VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Předzpracování obrazu pro indexování rozsáhlé kolekce dat

# Image Preprocessing for Large Scale Data Collections

2012 Martin Šurkovský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Martin Šurkovský**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Předzpracování obrazu pro indexování rozsáhle kolekce dat
Image Preprocessing for Larga Scale Data Collections

Zásady pro vypracování:

Cílem práce je sestavit aplikaci, pro předzpracování velké kolekce multimediálních obrazových dat. Práce bude obsahovat analýzu návrh a implementaci jádra aplikace včetně metod pro předzpracování a segmentaci obrazu. Podmínkou je univerzálnost řešení, počítající s doplňováním dalších algoritmů a postupů.

Aplikace
Aplikace bude tvořit nástroj pro testování algoritmů na zpracování obrazu. Jednotlivé algoritmy bude možné spojovat pomocí modulů do grafu, který určí posloupnost prováděných kroků. Některé části mohou být také zpracovávány paralelně a tvořit např. vystup pro jiný algoritmus, nebude
se jednat o triviální sekvenční graf.
- 40% diplomové práce.
- Implementační prostředí bude JAVA.
- Jádro aplikace by mělo být hotové ke konci osmého semestru.
- Aplikace bude navržena modulárně, tak aby ji bylo možné dále rozšiřovat (např. formou plug-inů nebo programovatelných modulů).
- K aplikaci bude dodána i analýza, návrh implementace a dokumentace.
- V dokumentaci podrobně popište všechny moduly zpracovávající obraz.

Algoritmy pro segmentaci obrazu
Tato část si klade za cíl nalézt vhodný postup segmentace obrazu. Jedná o kritickou část, která ovlivňuje kvalitu, rychlost a přesnost vyhledávaných výsledků.
- 60% diplomové práce.
- Zmapujte aktuální stav v oblasti segmentace obrazů.
- Dané algoritmy prostudujte a vyberte ty, které by byly vhodné pro účely rozpoznávání obrazů oběžných a pamětních mincí.
- S vybranou množinou algoritmů provádět testování, na výkon a přesnost. Pokud bude třeba, algoritmy optimalizujte pro daný problém nebo navrhněte vlastní řešení.
- Vybrané algoritmy otestujte na výkon a přesnost.

Řešení porovnejte s existujícími systémy.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radoslav Fasuga, Ph.D.**

Datum zadání: 19.11.2010

Datum odevzdání: 04.05.2012


doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2012

.....

Na tomto místě bych rád poděkoval vedoucímu práce Ing. Radoslavu Fasugovi, Ph.D nejen za vedení a cenné rady při tvorbě této práce, ale také za to, že mě zapojil do procesu vědeckého bádání, což vedlo k mému rozhodnutí dále pokračovat v navazujícím studiu. V neposlední řadě bych velice rád poděkoval své rodině a zejména rodičům, za jejich dlouhodobou podporu a důvěru v průběhu celého mého studia.

## Abstrakt

Tato diplomová práce se zaměřuje na problematiku zpracování obrazů mincí, pro účely jejich uložení a následného vyhledávání. Je rozčleněna do dvou hlavních částí. V první polovině je představen implementovaný nástroj, který by měl usnadnit modelování a testování nových komplexnějších metod, sestavených ze základních operací. V druhé části se nachází popis navržených metod, vč. dosažených výsledků. Většina metod je založena na principu normování natočení mince v obraze. Na začátku druhé poloviny jsou přestaveny přístupy, pomocí kterých toho lze dosáhnout. Všechny metody jsou vzájemně porovnány a je diskutováno, jak nastavení jednotlivých parametrů, některých klíčových operací ovlivňuje konečné výsledky vyhledávání. Úplně nakonec je prezentována metoda, založena na jiném principu popisu obrazu mince, který je nezávislý na jeho natočení.

**Klíčová slova:** Mince, vizuální programování, vyhledávání v obrazových datech, zpracování obrazu.

## Abstract

This master thesis is focused on the issue of image processing of coins with intention to save it into a databse with the possibility of the next searching. It's divided into two main parts. In the first part is introduced the implemented tool for modeling and testing of new more complex methods, which are composed from some basic operations. In the second part we can find description of proposed including results of made tests. Most methods are based on standardization rotation of coin. At the begining of second part are introduced approaches which can help to achieve it. All of methods are tested and compared to each other. It is conducted debate about, how the setting of parameters, from some key operations, influences results of searching. At the end is presented method, that is based on another approach which describes an image of coin independ on the his rotation.

**Keywords:** Coins, visual programming, search in the image data, image processing.

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| API | – | Application Programming Interface |
| DSS | – | Decentralized Software Services |
| FBP | – | Flow-based Programming |
| GUI | – | Graphical User Interface |
| OS | – | Operační systém |
| UML | – | Unified Modeling Language |
| VCL | – | Visual Component Library |
| VPL | – | Visual Programming Language |
| XML | – | eXtensible Markup Language |
| XSD | – | XML Schema Definition |

# Obsah

# Seznam tabulek

# Seznam obrázků

# Seznam výpisů zdrojového kódu

# 1 Úvod

Tato práce je již čtvrtou v pořadí v rámci projektu na rozpoznávání pamětních a oběžných mincí. První prací, která se touto tématikou začala, byla bakalářská práce [1] Ing. Petra Kašpara. V rámci ní vytvořil první verzi internetového katalogu, který umožňoval vcelku jednoduchou cestou, v malé kolekci (řádově stovky) dat, vyhledávat mince na základě jejich fotografie.

Na tuto práci jsem v minulosti navázal se svou bakalářskou prací [2], která se zaměřovala na problém natočení mince v obraze. Problém se tehdy podařilo částečně vyřešit a oproti původní práci, kdy bylo pro jednu minci ukládáno 72 vzorů[1], se tehdy novou metodou normování natočení, podařilo snížit počet rotací u stejné mince na 3 až 5 různých verzí. Snížení počtu ukládaných natočení otevřelo cestu pro indexaci mnohem větší kolekce obrázků mincí.

V minulém roce (2011) kolega Kašpar odevzdal diplomovou práci, ve které využil optimalizované metody normovaní natočení, tzv. metoda rozšířeného histogramu (kap. 6.1), navázal na svou bakalářskou práci a vytvořil novou verzi internetového katalogu mincí [2]. V něm je nyní za indexovaná kolekce cca. 140000 kusů mincí. Navíc umožňuje výměnu metody zpracovávající obrazy mincí, což přináší možnost s každou novou lepší metodou, aktualizovat vyhledávací část katalogu a zlepšovat tak výsledky vyhledávání. V práci se také testoval použitelnost různých druhů databází pro danou problematiku a různých způsobů ukládání dat.

Hlavním cílem této práce, je poskytnout komplexní nástroj pro snadné vytváření a testování nových metod zpracovávající obrazy mincí, a ne jen jich. Problém metod zpracovávající obraz je, že se skládají z několika dílčích operací, různě spolu komunikujícími a psát pro každý nový nápad novou verzi aplikace je neefektivní. Ve své bakalářské práci jsem tehdy vytvořil jednoduchý nástroj, který uměl jednotlivé operace řadit sériově za sebe. To se postupem času ukázalo jako ne zcela dostačující. Nový nástroj by měl umožnit jednotlivé operace seskupovat do složitějších struktur (orientovaných grafů). Navíc by měl být snadno rozšiřitelný o další moduly a společně s katalogem, poskytnout jednotnou vývojářskou platformu pro další studenty. Ti se budou moci zaměřit, již pouze na implementaci nových metod a nemuset programovat už hotové operace, neřešit problémy s komunikací, ukládáním, atp. Je totiž záměrem v projektu dále pokračovat, rozšířit jej i o rozpoznávání bankovek a bylo by žádoucí, aby další studenti mohli využit již hotových věcí. Nová aplikace by měla umožnit snadnější práci v týmu a urychlit a zefektivnit další vývoj.

Druhým z hlavních cílů je využít novou aplikaci a pokusit se vylepšit stávající metodu, popř. přijít s jiným přístupem, který by zlepšil výsledky vyhledávání. Kolega skončil ve své práci s celkovou úspěšností vyhledávání 80%. Je tak dále prostor pro další zlepšování.

[1] Pro každou minci byla vypočtena všechna natočení v rozmezí $5^\circ$.

[2] `http://identifycoin.vsb.cz`

## 1.1 Struktura práce

Práce začíná úvodem do historie původní aplikace, jsou definovány a rozebrány požadavky na novou verzi, plus je navržena její základní vizuální podoba.

V druhé kapitole je definován pojem vizuálního programování, jelikož nový nástroj je možné do této kategorie zařadit. Je zmíněna historie daného odvětví, vybráno a popsáno několik zástupců tohoto typu programů, od profesionálních a velmi drahých řešení až po open-source nástroje. Je také ve stručnosti zmíněn nástroj vyvíjený na katedře informatiky VŠB – TU Ostrava, který se zaměřuje na problematiku modelování paralelních úloh.

Ve třetí nejobsáhlejší části, je popis nově implementovaného nástroje. Na začátku je aplikace představena jako celek. Přičemž popis směřuje stále, k detailnějším informacím o tom, jak fungují její jednotlivé části a jak jsou implementovány. Ke konci kapitoly je popsán způsob, jakým lze aplikaci rozšiřovat o nové metody, či jak definovat novou lokalizaci.

Poslední část práce se věnuje popisu metod, použitých ke zpracování obrazu mincí. Je provedeno několik testů s různými modifikacemi původní metody, shrnuty jejich výsledky a diskutováno, jak jednotlivá nastavení ovlivňují finální výsledky vyhledávání. Nakonec je představena metoda založena na jiném principu, která se místo normování rotace, pokouší popsat minci nezávisle na jejím natočení. Vzhledem k množství prováděných testů, byla pokaždé indexována kolekce 25000 náhodně vybraných kusů, namísto celé původní kolekce.

# 2 Aplikace

Tato část tvoří úvod k implementované aplikaci. Je zde zmíněna její historie, důvody a motivace které vedly k implementaci nové verze. Následuje specifikace základních požadavků, jejich stručná diskuze a hrubý návrh uživatelského rozhraní. Z toho všeho vyplyne, že výsledný nástroj spadá do kategorie, tzv. *vizuálního programování*. To co se myslí pod pojmem vizuálního programování a jaký je aktuální stav v této oblasti blíže pojednává následující kapitola (kap. 3).

V kapitole 4 již následuje detailní rozbor implementovaného nástroje, od celkového pohledu na systém, až po analýzu jednotlivých modulů. Jsou definovány možnosti a omezení jeho použití. Na závěr je provedeno srovnání s existujícími řešeními, diskuze nad tím v čem jsou lepší, v čem by naopak mohla být horší a co přináší implementovaný nástroj nového, resp. v čem se výrazně od stávajících systémů liší.

## 2.1 Historie

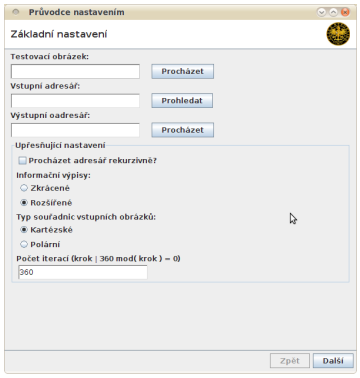
Práce navazuje na bakalářskou práci [2] a nová aplikace je vlastně druhou verzí původního nástroje. K implementaci zcela nového nástroje se přistoupilo z důvodu postupné změny požadavků na to co by měla aplikace umět a na to jak by se s ní mělo pracovat.

Hlavním nedostatkem původní verze bylo pouhé jednoduché sériové řazení jednotlivých operací za sebe. To vycházelo z „prapůvodní“ verze kdy aplikace měla sloužit pro práci s obrázky v konzolovém režimu na serveru, kde jednotlivé parametry tvořily metody se svými argumenty. Vzhledem k řešenému problému, počtu metod a jejich parametrů, který se postupně navyšoval se přistoupilo k vytvoření grafické nadstavby pro snazší sestavení výsledného postupu. Několik ukázek z dané aplikace je možné vidět na následujících obrázcích (obrázky 1 až 3).

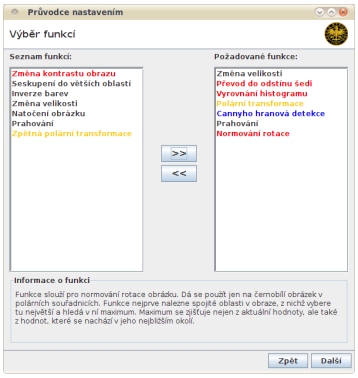
Jak je z obrázků vidět aplikace se sestávala z několika kroků, které byly naskládány za sebe ve formě průvodce. Za prvé bylo třeba zadat inicializační parametry, jako adresáře se vstupními a výstupními daty, plus další dodatečné informace (obr. 1). Následovaný asi nejdůležitějším krokem, sestavení výsledného postupu z dílčích metod (obr. 2). Nakonec se nastavily parametry vybraných metod, např. na posledním obrázku je vidět okno pro nastavení Cannyho hranového detektoru (obr. 3).

Druhým velkým nedostatkem bylo nepříliš snadné rozšíření o další metody. Bylo to dáno tím, že pro přidání nové metody se musel napsat ne jen samotný algoritmus, ale také vizuální komponenta sloužící pro nastavení všech jeho parametrů (viz. obr. 3). Nakonec se to muselo zakomponovat do celého nástroje a aplikaci nově zkompilovat. Tento fakt by ovšem velice znesnadňoval možnou budoucí spolupráci s dalšími vývojáři.

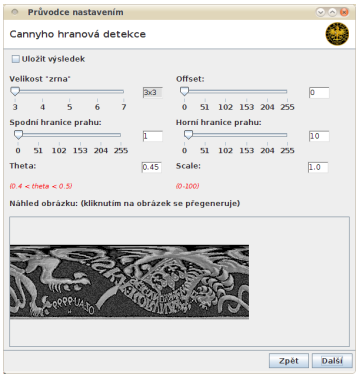
Pro nás toto řešení bylo více, či méně dostačující, ale ke konci se začaly projevovat nedostatky hlavně se sériovým zpracováním operací. Proto jsme přistoupily k implementaci nové aplikace založené na zcela jiném přístupu. Nicméně z původní verze zůstalo to nejdůležitější, čímž jsou samotné metody na zpracování obrazu.




Obrázek 1: Obrazovka s prvotním nastavením.




Obrázek 2: Výběr posloupnosti funkcí.




Obrázek 3: Ukázka nastavení jedné z funkcí.

## 2.2 Motivace

Důvody pro naprogramování nové verze vyplývají z velké části z nedostatků popsaných v historii. Ovšem jednou z největších motivací je zcela jiné základní použití aplikace.

Předchozí verze byla navržena pro sestavení výsledného postupu z jednodušších metod. Přičemž se skrytě na začátku předpokládalo, že daný postup bude „jednoduchý“. I vzhledem k tomu, že se jedná o zpracování obrazu myslelo se, že bude stačit jakási postupná aplikace několika málo metod na vstupní obrázek. Podobně jako například u maker v Photoshopu[3]. Tento předpoklad se ovšem ukázal být chybným a ve skutečnosti docházelo a dochází k případům, kdy je třeba se vrátit, či získat data z metody, na jejíž výstup už byla aplikována jiná operace.

Druhou motivací je, že projekt na rozpoznávání a identifikaci mincí se stal docela rozsáhlým, na který bude možné v budoucnu určitě navázat, plus navíc je v současné době rozpracován projekt na rozpoznávání bankovek. Nový nástroj si pak klade za cíl společně s katalogem[4] vytvořit jednotné prostředí pro další vývoj tak, aby další nemusely začínat od začátku, ale pokračovat v již započaté práci a více se soustředit na jednotlivé algoritmy a možné přístupy. Zmiňovaný katalog je součást diplomové práce [3] kolegy Petra Kašpara.

## 2.3 Seznam požadavků

Předchozí řešení napomohlo identifikovat slabá místa a vlastně i použití a účel aplikace. Nástroj se daleko více posune k uživateli tak, aby měl důvod s ním spolupracovat a zároveň, aby nebyl zbytečně složitý. Respektive aby byl pro uživatele co možná jak nejjednodušší. **Jednoduchost** je vlastně jeden ze základních požadavků, který se potáhne celou aplikací, jako taková červená nit'. Následující výčet blíže definuje požadavky, kladené na výsledné řešení:

[3] `http://success.adobe.com/cs/cz/sem/products/photoshop.html`
[4] `http://identifycoin.vsb.cz/`

1. Uživatelská přívětivost,
2. snadná ovladatelnost,
3. jednoduchá rozšiřitelnost,
4. možnost paralelního zpracování,
5. modulární implementace,
6. multiplatformní řešení,
7. zachování možnosti spouštět výsledné postupy z příkazové řádky.

## 2.4 Rozbor požadavků a návrh řešení

Vzhledem k požadavkům kladených na aplikaci bylo rozhodnuto, že zřejmě nejlepší způsob jak modelovat složitější algoritmy bude formou nějakého orientovaného grafu. To znamená že jednotlivé operace budou reprezentovat uzly grafu a hrany znázorňovat komunikaci mezi jejich vstupy a výstupy. Navíc využití orientovaného grafu pro modelování algoritmů si více, či méně vynutí dodržení většiny bodů ze seznamu požadavků.

**Uživatelská přívětivost,** je jedním ze základních kamenů, protože s aplikací by měli spolupracovat další uživatelé a programátoři. Ani jednu z těchto skupin by aplikace neměla odrazovat, ale naopak nabízet takový komfort, aby jim ulehčila práci a chtěli s ní pracovat. Z pohledu uživatele by nástroj neměl klást přehnané požadavky na něj a člověk by se měl po chvíli práce v ní nebo přečtení stručného návodu snadno zorientovat. Využití orientovaného grafu tomuto požadavku vcelku nahrává. Jelikož uživatel dostane k dispozici seznam dostupných operací, z nichž si vybere ty, které potřebuje a pouhým zakreslením komunikace mezi moduly definuje výsledný proces (algoritmus).

**Jednoduchá rozšiřitelnost,** je naopak spíše otázkou toho, co vše bude vyžadováno po programátorovi, aby naprogramoval pro přidání nového modulu (operace) do aplikace. V rámci zachování jednoduchosti samozřejmě co nejméně, tedy to co je nezbytně nutné. Na druhou stranu jednotlivé moduly mohou být vytvořeny různými programátory a programátor a uživatel nemusí být vždy jedna a ta samá osoba. Proto i pro uživatele musí existovat jednoduchá cesta jak přidat již hotový modul.

**Možnost paralelního zpracování,** když už bude použít pro návrh a modelování komplexních algoritmů orientovaný graf, byla by škoda nevyužít jeho přirozených vlastností a tam kde to bude možné nechat některé operace běžet paralelně.

**Modulárnost řešení,** ta už vyplývá částečně z jednoduché rozšiřitelnosti, avšak i celá aplikace by měla být tvořena z co největšího množství samostatných modulů. Je to dáno tím, že ne vše se dá ze začátku předpovědět a udělat dokonale. Pokud by se v budoucnu ukázalo, že některá část aplikace nevyhovuje, měla by být její výměna

co jak nejjednodušší. Tedy bez nutnosti přepisovat množství kódu, který přímo z danou části nesouvisí.

**Multiplatformní řešení,** se myslí taková aplikace, kterou bude možné spustit na různých operačních systémech. Z toho to důvodu byl zvolen programovací jazyk Java, který by měl usnadnit správu aplikace pro různé systémy. Multiplatformnost se ovšem týká pouze nástroje jako takového. Tento požadavek již nemůže být zaručen u rozšiřujících modulů dodaných třetí stranou. Pokud například autor daného modulu využívá některé funkcionality specifické pro danou platformu nebo využívá prostředků, které je třeba nutné doinstalovat,[5] pak nástroj nemůže za to, že ne všechny moduly bude možné využít na kterékoli platformě.

**Zachování možnosti spouštět namodelované postupy z příkazové řádky,** přináší možnost spouštět hotová řešení na serveru bez grafického rozhraní, ovšem navržené postupy bude možné povětšinou pouze spouštět. Možnost editace bez grafické nadstavby bude značně omezena. Důvodem zachování této funkcionality je, že aplikace by měla komunikovat i s katalogem pamětních a oběžných mincí [3], který běží právě na serveru bez grafického rozhraní.

### 2.4.1 Hrubý návrh uživatelského rozhraní

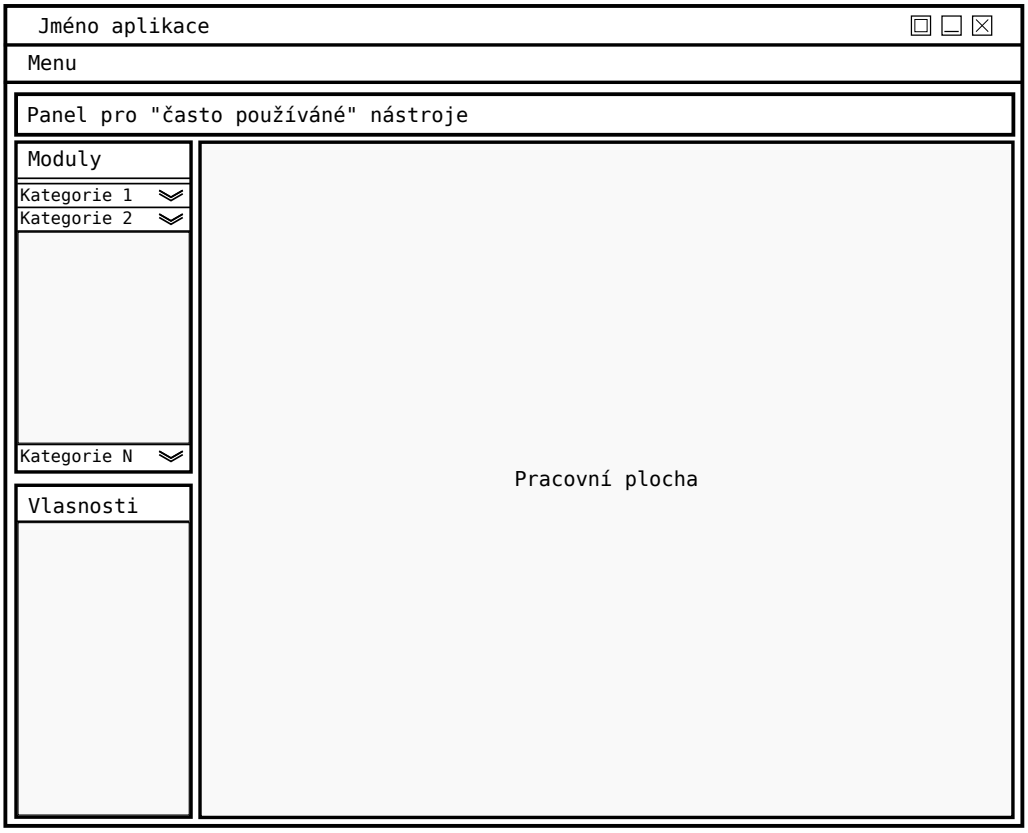
Z rozboru všech požadavků definitivně vyplynulo, že pro modelování algoritmů sestavených z dílčích modulů bude použito některé z forem orientovaného grafu. Nástrojů pro kreslení grafů či různých technických schémat v dnešní době existuje nepřeberné množství. Takovým příkladem mohou být například nástroje typu *MS Visio* nebo různé formy case nástrojů pro práci s UML diagramy. V aplikaci vyvinuté v rámci této práce půjde prakticky o něco podobného. Základní vzhled je tak do značné míry inspirován těmito nástroji.

Hrubý návrh uživatelského rozhraní je možné vidět na obrázku 4. Je rozdělen do čtyřech hlavních částí. V horní části bude menu s panelem tlačítek často používaných funkcí. Ve středu zabírá největší část pracovní plocha, v rámci níž bude možné modelovat výsledné algoritmy. Po boku na levé straně se bude panel s dostupnými operacemi, jež bude možné řadit do různých kategorií. A nakonec se zde bude nacházet také panel s vlastnostmi jednotlivých operací, které pomocí něj bude možné měnit. První návrhy a nápady na aplikaci pochází z roku 2010 a jsou částečně formulovány v [4].

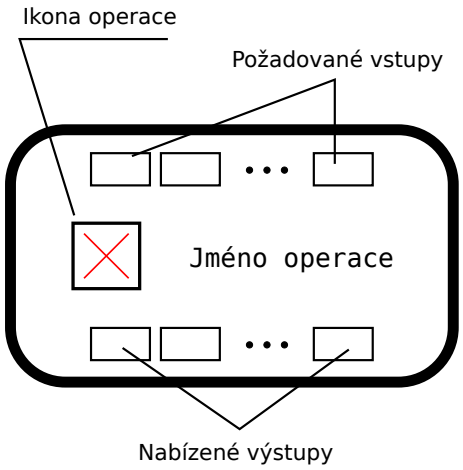
Dále tím, že nástroj bude pracovat s jednotlivými algoritmy, kterých bude možné „natáhnout“ na pracovní plochu, je třeba nějak definovat jejich vizuální podobu. Ta by měla zachycovat všechny důležité informace tak, aby bylo možné s metodou rozumně používat. Takové informace jsou bezesporu definované vstupy a výstupy a jméno operace. Navíc je přidána ikona pro každou operaci pro snazší rozlišení operací na pracovní ploše. Návrh vizuální reprezentace operace je vidět na obrázku 5.

Nástroj ovšem nebude pouze nějakým „kreslítkem“, ale umožní i navržené algoritmy spouštět. S tímto vším dohromady spadá aplikace do kategorie nástrojů pro takzvané

[5] Typicky tím může být např. databáze.




Obrázek 4: Základní návrh kostry aplikace.




Obrázek 5: Základní návrh vizuální reprezentace operace.

vizuální programování. O tom, oč se vlastně jedná a jaký je aktuální stav v této oblasti, pojednává následující kapitola.

# 3 Aktuální stav v oblasti vizuálního programování

Tato část se pokusí dát odpověď na otázku, co je to vizuální programování a vizuální programovací jazyky. Na začátku bude trocha z historie této techniky, budou řečený důvody, které vedly jejímu vzniku. Dále se pokračuje faktickým popisem toho co je a co není vizuální programování, s krátkým zamyšlením se nad možnostmi různého chápání toho pojmu. Následuje seznam vybraných zástupců aktuálně dostupných řešení. U každého z nich je stručný popis a náhled jak dané prostředí vypadá. Nakonec je diskutováno využití těchto nástrojů v dnešní době s úvahou nad jejich možnostmi využití v budoucnu.

## 3.1 Historie

Historie tohoto odvětví je stará již více než třicet let [6]. Její začátky sahají do počátku osmdesátých let minulého století. Ovšem prvotní myšlenky jsou ještě o něco starší. Původní motivací bylo využití pro masivní paralelizaci úloh [7], avšak pro paralelní počítání nebyla zrovna vhodná von Neunmannova architektura počítače. V druhé polovině sedmdesátých let tak byl představen alternativní návrh, tzv. dataflow architektura. Dataflow architektura se liší v tom, že obsahuje pouze lokální paměti a operace provádí hned, jakmile má k dispozici data. Od těchto myšlenek již není daleko ke skutečným vizuálním programovacím jazykům. Formulaci toho, co by jazyk pro takovou architekturu měl splňovat, sepsal W.B. Ackerman ve svém článku: *Data flow languages* [8]. Jedním z vizuálních programovacích jazyků, který začal vznikat v první polovině osmdesátých let a vyvíjí se až do dnešní doby je LabVIEW (viz. kap. 3.3.2).

## 3.2 Vizuální programování

Existuje mnoho způsobů jak chápat pojem vizuálního programování. Takové intuitivní vysvětlení by mohlo být, že se jedná o techniku, kdy uživatel vytváří či upravuje program manipulací s grafickými prvky, reprezentující určité části programu. To jak velkou část kódu jednotlivé grafické prvky reprezentují, záleží na stupni abstrakce daného prostředí. Může se jednat o objekty na nízké úrovní, představující primitivní typy, podmínky, cykly, atp. nebo naopak o komplexní funkce, které komunikují s okolím pomocí svých vstupů a výstupů.

Definice vizuálního programovacího jazyka (VPL) je již na druhou stranu trochu více jasná a dává podklad pro to, jak oddělit co do dané kategorie spadá, a co ne.

**Definice 3.1 (VPL [9])** *Vizuální programovací jazyk, je takový jazyk, který umožňuje specifikovat program pomocí dvou nebo více dimenzí.*

Ovšem i tak může pojem působit zmatky. Ukázkou je například prostředí pro definici uživatelského rozhraní, kupříkladu *JBuilder*. Otázka zní: Jedná se o vizuální programovací jazyk, a nebo ne? Odpověď je **ne**. Protože se stále jedná o textový jazyk, který akorát používá grafický nástroj pro snazší zápis kódu.

Vizuální programovací jazyky lze klasifikovat podle mnoha kriterií. Tyto kriteria byly sepsána v článku *A Classification System for Visual Programming Languages* [10]. Takové nejvíce říkající rozdělení by mohlo být podle vizuální reprezentace. Jazyky nebo lépe řečeno jejich prostředí lze tak rozdělit do dvou základních skupin:

- na ty založené na tabulkách
- nebo na ty založené na různých formách diagramů.

Příkladem první skupiny je typicky nějaký tabulkový procesor, např. MS Excel nebo Calc[6]. Příkladem druhé skupiny může být kupříkladu aplikace řešená v rámci této práce nebo některé další programy. V následující části je několik vybraných reprezentantů této kategorie stručně popsáno.

Na závěr ještě zdůvodnění, proč nepatří klasické textové jazyky do VPL. Protože tyto jazyky nejsou dvou-dimenzionální, jelikož překladač nebo interpret je zpracovává jako jednorozměrný tok dat.

## 3.3 Existující nástroje

Následující seznam představuje několik vybraných nástrojů, zabývajících se touto problematikou. Výčet si neklade za cíl postihnout vše co existuje, ani to není možné. Například pouze na wikipedii je seznam cca. stovky aplikací. Seznam má spíše představit některé dané nástroje, říci k čemu slouží a co nabízejí. Jsou v něm zahrnuti zástupci od velkých komerčních řešení, typu all-in-one, až po menší, více specifiky zaměřené projekty.

### 3.3.1 Agilent VEE

`http://www.agilent.com/find/vee`

Agilent VEE neboli **V**isual **E**ngineering **E**nvironment je grafické programovací prostředí optimalizované pro použití s elektronickými přístroji. Jedná se o profesionální plnohodnotný vizuální programovací nástroj, který je založen na paradigmatu *dataflow programming*.

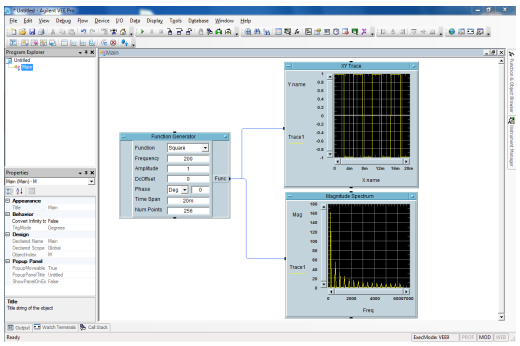
Je určen pro platformu MS Windows, běží na .NET frameworku a celkově velmi dobře spolupracuje s vývojovými nástroji microsoftu. Také umožňuje spolupraci s Matlabem, a to tak že je možné v jednotlivých blocích přímo volat matlabovské funkce. Nicméně největší síla nástroje spočívá v tom že umí komunikovat s externími elektronickými zařízeními, měřícími přístroji, atd. Měl by lidem pomoci s tvorbou programů pro zpracování výsledků měření, jejich vizualizaci či samotnou simulaci měření.

Jak aplikace zhruba vypadá je možné vidět na dvou následujících obrázcích. Prostředí je značně podobné klasickým vývojovým prostředím, akorát místo části s editací zdrojového kódu je pracovní plocha kde jsou vkládány jednotlivé komponenty.

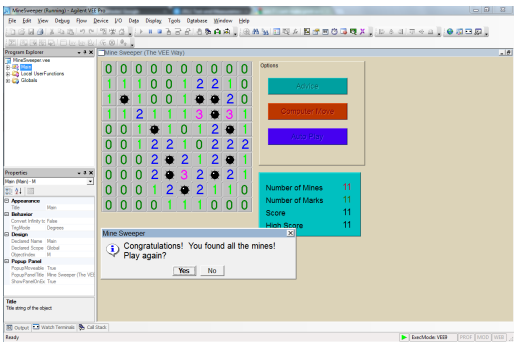
První obrázek (obr. 6) obsahuje jednoduchý příklad z tutoriálu, který ukazuje nějakou základní práci se signály. Na druhém obrázku (obr. 7) je vidět že možnosti použití se

[6] `http://www.openoffice.cz/calc`

meze nekladou a pomocí tohoto nástroje je možné vytvářet i „standardní“ aplikace jakými mohou být například hry.



Obrázek 6: Příklad jednoduchého programu.



Obrázek 7: Ukázka aplikace - hledání min.

Základní pohled na to jak nástroj pracuje je takový, že jednotlivé bloky reprezentují akce které je možné spouštět. Každý blok má několik druhů pinů, tj. rozhraní pomocí kterých blok komunikuje se svým okolím. Těmi nejdůležitějšími jsou vstupní a výstupní piny. Jednotlivé bloky je možné následně propojovat mezi sebou a toto propojení reprezentuje přenášená data. Takto vytvořený obrázek pak tvoří hotový program, který je možné rovnou spustit.

Do Agilent VEE je možné přidávat i vlastní funkce či uživatelské objekty. První způsob je vytvořit nový uživatelský objekt přímo v prostředí, definují se jeho vstupy, výstupy a vnitřní část je opět tvořen grafickými komponentami. Druhým způsobem je naprogramovat komponentu v některém z klasických textových programovacích jazyků. Agilent VEE podporuje Visual Studio .NET assemblies, takže je možné vyžít jakýkoliv jazyk, který je dostupný ve Visual Studiu.

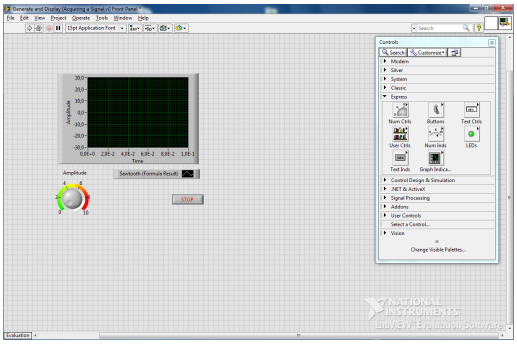
**Zhodnocení:** jedná se o profesionální nástroj pro vizuální programování, čemuž odpovídá i cena, která činí $1836 za licenci. Subjektivně nástroj nepatří zrovna mezi user friendly aplikace. Chce to svůj čas se s ním naučit pracovat, ovšem když to člověk zvládne věřím že to může zrychlit a ulehčit práci.

### 3.3.2 LabVIEW

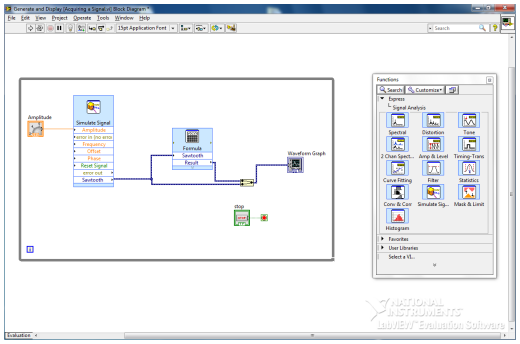
`http://www.ni.com/labview`

Název aplikace LabVIEW je zkratkou pro **Lab**oraty **V**irtual **I**nstrumentation **E**ngineering **W**orkbench. Jak název napovídá jedná se o jakousi virtuální laboratoř umožnující virtuálně pracovat s různými elektronickými zařízeními. Ovšem to není všechno ve skutečnosti se jedná opět o plnohodnotný vizuální programovací jazyk, jehož vývoj započal v roce 1983. První verze vyšla o tři roky později, tehdy pro Apple Macintosh [11]. V dnešní době se jedná o komplexní profesionální nástroj s obrovským množstvím rozšiřujících knihoven.

Jazyk který LabVIEW používá se nazývá *G* a je opět založen na paradigmatu dataflow programming. Vývojové prostředí se skládá ze dvou hlavních oken. První slouží pro tvorbu uživatelského rozhraní (obr. 8) a do druhého se kreslí blokový diagram popisující chování aplikace (obr. 9). Vývoj uživatelského rozhraní a logické části aplikace je tak svázán dohromady.



Obrázek 8: Definice uživatelského rozhraní.



Obrázek 9: Okno s blokovým diagramem.

Základní jednotkou či komponentou je vizuální zařízení (visual instrument - VI). Jedná se vlastně o funkční programy nebo malé rutiny, které se využívají jako komponenty ve složitějších systémech. Plně modulární charakter zajišťuje znovupoužitelnost hotových programů v jiných projektech bez nutnosti dalších úprav.

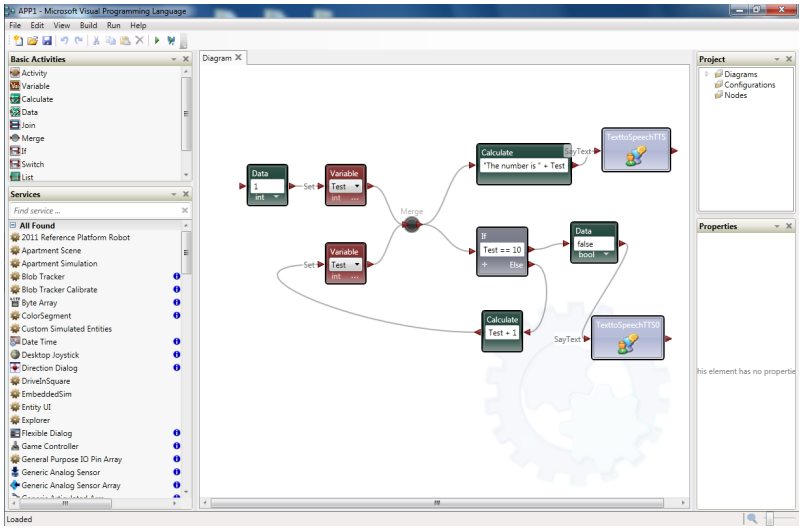
Velkou výhodou aplikace je podpora široké škály ovladačů pro přístup k hardwarovým zařízením. Navíc již zmíněné velké množství knihoven s matematickými funkcemi, statistickými, generátory signálu, zpracování signálu, ale i třeba rozsáhlou knihovnou pro tvorbu aplikací zabývající se strojovým viděním. Nástroj je plně podporován na systému MS Windows, částečně pak na Mac OS X a Linuxu (Red Hat Enterprise, SUSE). Na zbývajících dvou platformách nemusí být dostupné všechny ovladače a rozšiřující moduly.

**Zhodnocení:** nástroj je obdobou předchozí aplikace, ovšem osobně jej hodnotím jako lepší a propracovanější verzi. Hlavně co se týče tvorby programu oddělením vizuální a logické části napomáhá k lepšímu porozumění diagramu a i po vizuální stránce se zdá být zdařilejším. Na druhou stranu se nejedná vůbec o levnou záležitost. Sice je k dispozici několik druhů licencí, rozdělených podle způsobu použití a možnostmi které nástroj nabízí. Ale i tak cena za základní verzi je 29900 Kč. Verze professional pak vyjde na 106900 Kč. Navíc za velkou většinu rozšiřujících modulů se také platí. Ceny modulů se pohybují v řádech od jednotek až po několik stovek tisíc.

### 3.3.3 Microsoft VPL

`http://msdn.microsoft.com/en-us/library/bb483088.aspx`

Dalším příkladem z oblasti dataflow programming je Microsoft Visual Programming Language, který je součástí jejich Robotics Developer Studia. Nástroj je určen pro začínající programátory, kteří mají základní povědomí o programování a pro programátory robotů, pro které nabízí množství již hotových funkcí. Ve srovnání s předchozími prostředími je nástroj opravdu jednoduchý a člověk se v něm relativně snadno zorientuje. Prostředí je rozděleno do několika částí, ve středu je hlavní pracovní panel a na levé straně dva panely s dostupnými bloky, pomocí nichž se sestavuje výsledný program. Jak prostředí vypadá je možné vidět na obrázku 10.




Obrázek 10: Microsoft Robotics Developer Studio.

Stavebním kamenem jsou bloky (aktivity), které je možné dále spojovat do složitějších celků. Základní aktivity které je možné použít jsou bloky pro definici dat (*data*), řízení toků (*if* a *switch*), kombinaci zpráv (*join* a *merge*), provádění výpočtů (*calculate*) a uložení proměnné (*variable*). Plus navíc je definována aktivita, která obsahuje nějaký diagram. Tak je možné shlukovat některé celky do samostatných aktivit a znovu je použít. Jedná se o jeden ze způsobů jak definovat vlastí bloky. Druhou možností je využít DSS services. Jedná se vlastně o zkompilovaný kód napsaný v C# využívající vcelku jednoduché komunikační rozhraní. Všechny bloky nacházející se ve spodní části levého panelu jsou takto naprogramovány. Díky podpoře asynchronního volání v DSS je také možné výsledný program provádět paralelně.

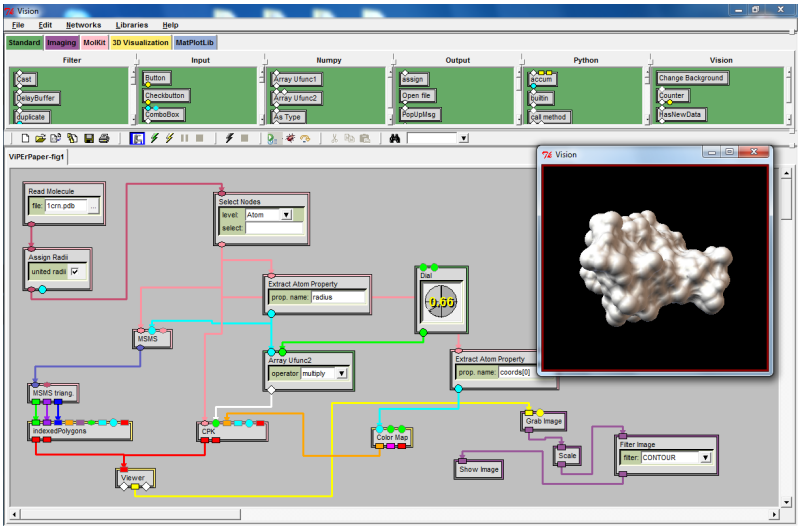
**Zhodnocení:** jedná se o opravdu jednoduchý a snadno pochopitelný nástroj, což hodnotím oproti předchozím dvěma prostředím jako bonus. Také to k čemu je určen a tedy programování robotů si myslím že může ulehčit práci. Navíc s širokou paletou již hotových funkcí může v celku jednoduše vytvořit program pro robota i méně zkušený programátor. Co už se mi moc použitelné nezdá je pokusit se vytvořit program pouze ze základních komponent. Zde vidím využití jako demonstrační nástroj pro výuku programování. V

této situaci je efektivnější napsat zdrojový kód v některém z klasických programovacích jazyků. Na závěr mile překvapilo zjištění, že nástroj je bezplatně[7] k dispozici jak pro komerční tak nekomerční využití.

### 3.3.4 Vision

`http://mgltools.scripps.edu/packages/vision`

Jedná se o další vizuální programovací prostředí, ve kterém mohou uživatelé pomocí modulů interaktivně vytvářet síť popisující novou kombinaci výpočetních metod, bez nutnosti psát kód. Nástroj je součástí balíku MGLTools, který se zaměřuje na vizualizaci a analýzu molekulárních struktur. Vision [12] je napsán v Pythonu a TCL/TK.




Obrázek 11: Ukázka nástroje Vision.

Základem jsou opět moduly, které je možné mezi sebou propojovat. V tomto případě tvoří moduly jen jakýsi obal pro funkcionalitu, která je jinak dostupná v Pythonu, C nebo Fortranu. Tento fakt umožňuje aplikaci rozšiřovat o již hotové funkce a knihovny napsané v daných jazycích. Nástroj tak obsahuje například knihovnu s moduly pracujícími s OpenGL. Dále jsou dostupné hlavně knihovny pro práci s daty reprezentujícími molekulární struktury a z pohledu této práce mě osobně zaujala knihovna Imaging pro práci s obrázky. Ta ovšem obsahuje základní kolekci funkci, jako jsou načtení a uložení obrázku, aplikace filtrů a základní geometrické transformace.

**Zhodnocení:** jedná se o celkem zajímavý nástroj, který je dostupný na všechny základní platformy. Jako takový ho je možné použít i pro komerční účely, ovšem ne tak všechny jeho knihovny. Co je v celku zajímavé, je to že nástroj po vizuální stránce působí jako by se

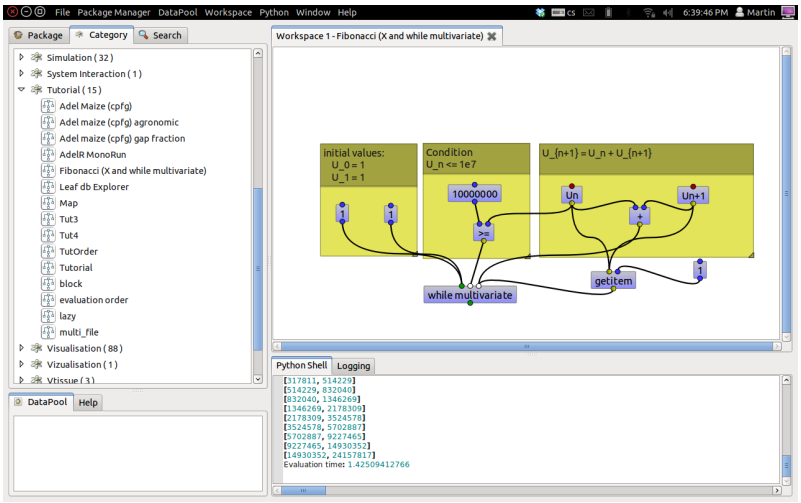
[7] `http://msdn.microsoft.com/en-us/robotics/aa731520`

jednalo o už nevyvíjený projekt někdy z druhé poloviny devadesátých let, ale skutečnost je přesně opačná. V době psaní této práce je poslední dostupná verze z února 2012.

### 3.3.5 OpenAlea

`http://openalea.gforge.inria.fr`

OpenAlea [13] je open-source software napsaný v Pythonu, který je určen pro modelování rostlin. Obsahuje moduly pro analýzu, vizualizaci a model růstu rostlin. Je možné doprogramovat vlastní moduly. K dispozici jsou jazyky C, C++, Python a Fortran. Na následujícím obrázku (obr. 12) je možné vidět jak prostředí vypadá. Základní rozvržení je velmi podobné jako v předchozích aplikacích.




Obrázek 12: Vizuální prostředí systému OpenAlea.

**Zhodnocení:** jedná se na první pohled o vcelku zajímavý nástroj, bohužel po instalaci uživatele překvapí spousta nepříjemných překvapení. Prvně v panelu s moduly je po instalaci nepřeberné množství různých funkcí. Ty jsou sice nějak kategorizovány, ale jejich pojmenování v mnoha případech vypadá jako jména získaná přímo ze zdrojového kódu. Sice jsou k nim k dispozici krátké popisky, ale ani ty někdy nepomohou. Druhým nedostatkem je že spousta ukázek a demo programů obsažených v aplikaci po instalaci nelze spustit. Hlavně takové ty zajímavější, které autoři uvádějí na webu projektu. Jedním z mála programů, který se mi podařilo spustit po instalaci je výpočet posloupnosti Fibonacciho čísel, jehož diagram je i na ukázkovém obrázku (obr. 12).

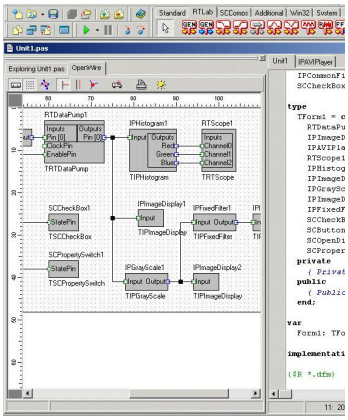
### 3.3.6 OpenWire

`http://www.mitov.com/products/openwire`

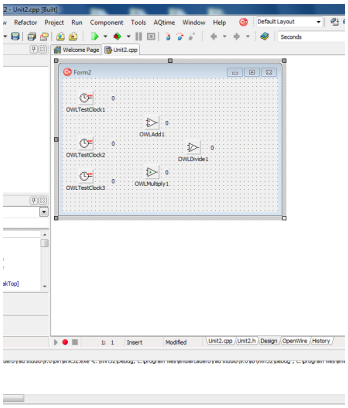
Jedná se o open-source knihovnu rozšiřující funkce Delphi nebo C++ Builderu. Jejím hlavním cílem je poskytnout relativně jednoduchou cestu jak přenášet data mezi rozdílnými VCL nebo Firemonkey komponentami.

Základem jsou opět bloky, které mohou obsahovat jeden čí více vstupů a výstupů. Vstupy a výstupy jsou v terminologii OpenWire nazývaný piny. Existují tři druhy pinů: vstupní, výstupní piny pro přenos dat a stavový pin, který je v knihovně od verze 2. Stavové piny jsou primárně navrženy pro výměnu informací o stavu komponent. Propojování mezi vstupními a výstupními piny je možné pouze v případě, že jsou datově kompatibilní. Piny totiž nesou v sobě informaci o typu dat, pro který jsou určeny.

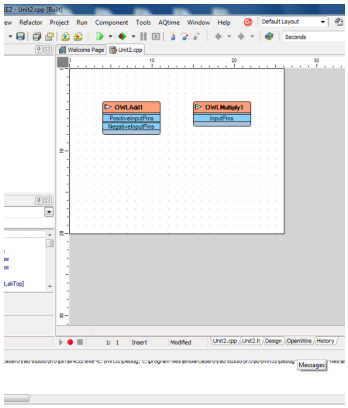
Ke knihovně jsou dostupné i rozšíření určená pro různé specifické oblasti použití. Pro nekomerční použití je možné přidat: *AudioLab*, *VideoLab*, *VisionLab*, *InstrumentLab*, *IntelligenceLab*, *SignalLab a PlotLab*. Ukázka ja vypadá knihovna OpenWire instalována do prostředí Delphi 7 je možné vidět na následujícím obrázku[8] (obr. 13).



Obrázek 13: Grafický editor OpenWire ve vývojovém prostředí Delphi 7.



Obrázek 14: Ukázkový příklad z tutoriálu, jednoduchý aritmetický výpočet.



Obrázek 15: Reálný stav OpenWire editoru.

**Zhodnocení:** Ač se jedná o open-source projekt je nutné mít pro jeho použití nainstalován buď *C++ Builde XE2* nebo *Delphi XE2*. Obě tyto vývojová prostředí jsou však již placená a jejich cena se pohybuje od €199 za verzi starter až po €3499 za plnou verzi.[9]

Nevýhodou je, že to co je vidět na obrázku 13 a ostatně i na oficiálních stránkách projektu je pouze ukázka toho, jak by to mělo v budoucnu celé fungovat. OpenWire editor se totiž stále vyvíjí. Nyní pokud uživatel nahraje knihovnu do jednoho z vývojových prostředí, tak má sice k dispozici dané komponenty, které může vkládat na okno tvořící program (builder grafického uživatelského rozhraní), ale už nemá možnost vizuálně programovat, tak jak to naznačuje obrázek 13 a ukázky na oficiálních stránkách. Příklad jednoduchého programu z tutoriálu s využitím OpenWire komponent je vidět na obrázku 14. To co je následně možné vidět v OpenWire editoru ukazuje obrázek 15. Editor nyní zobrazí pouze některé z objektů použitých v okně s návrhem aplikace a možnost propojovat mezi sebou jednotlivé vstupy a výstupy zatím nelze. Jediný způsob jak nyní piny propojit je v tabulce s vlastnostmi daného objektu, kde se ke konkrétnímu pinu vybere jeho protějšek ze seznamu všech použitých pinů.

[8] Zdroj: `http://en.wikipedia.org/wiki/File:OpenWireEditorD7.jpg`

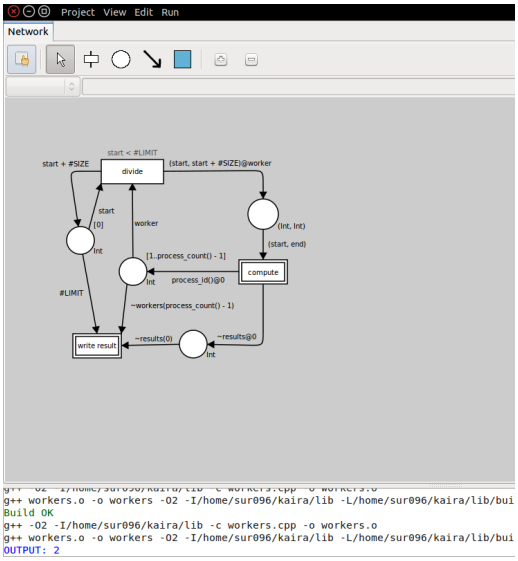
[9] Ceny z oficiálního e-shopu `https://store.embarcadero.com`

Co hodnotím ovšem velmi negativně, je fakt, že informace o tom, že se jedná pouze o ukázku editoru, který není plně funkční, je uvedena jednou větou v manuálu. Celý web projektu ovšem působí tak, že by vše mělo fungovat.[10]

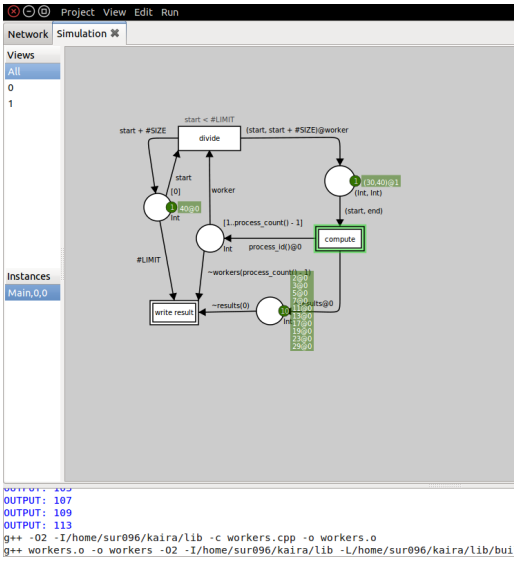
### 3.3.7 Kaira

`http://verif.cs.vsb.cz/kaira`

Jedná se o nástroj vyvíjený na katedře informatiky VŠB – TUO, jehož hlavním autorem je Ing. Stanislav Böhm. Kaira [14] je vizuální vývojové prostředí pro tvorbu paralelních aplikací, založené na barevných Petriho sítí [15]. Nástroj cílí na uživatelé, kteří by chtěli nějakým jednoduchým způsobem vytvořit paralelní verzi svojí aplikace a v danou chvíli to buď neumějí nebo nechtějí, např. kvůli složitosti. Příkladem takového použití může být paralelizace algoritmu ant colony optimization [16], kdy byla k dispozici sériová implementace a bylo třeba ji nějak jednoduše paralelizovat.




Obrázek 16: Kaira - prostředí pro definici sítě.




Obrázek 17: Kaira - simulace běhu sítě.

Kaira se programuje tak, že se vytvoří síť, která popisuje komunikaci v modelu. Takovouto síť je možné vidět na obrázku 16. Jedná se o jednoduchý program z dostupných ukázkových příkladů, kdy je třeba rozdělit nějaký úkol na několik dílčích úkolů, ty jednotně zpracovat a výsledek dát zase dohromady.

Kolečka na obrázku reprezentují místa, to jsou zpracovávaná data a obdélníky přechody, což jsou funkce, které data zpracovávají. Přechody v Kaiře mohou obsahovat libovolný kód (prozatím C/C++). Takto namodelovaný program je následně možné odsimulovat (obr. 17), popř. zkompilovat a dále spouštět již samostatně bez nutnosti použít Kairu.

[10] Testováno 17.3.2012, kdy poslední aktuální verze knihovny je z října 2011.

## 3.4 Souhrn

Vizuální programovací jazyky tvoří zajímavou alternativu přístupu k programování, ale jejich využití není vhodné na všechno. Textový popis je vlastně pro většinu úkolů stále lepší variantou, o čemž se zmiňuje i tzv. Deutsch limit [17]. Například nevhodné použití této techniky je pokoušet se pomocí ní naprogramovat obyčejný třídící algoritmus či pouhý průchod cyklem. Pro tyto situace a mnohé další je pořad nejefektivnější použití klasického textového programovaní. I toto může být důvod, proč se může zdát, že vývoj v této oblasti se na delší dobu ustálil. Největší boom výzkumu a vývoje byl od počátku osmdesátých let minulého století, až po zhruba první polovinu let devadesátých. Z té doby pochází nejvíce článků na toto téma a mnoho různých aplikací. Od druhé poloviny devadesátých let pak zájem o danou problematiku klesal.

Ovšem v posledních letech, zhruba od roku 2006, se zdá, že se vizuálního programování opět dostává do popředí a vyvíjí se nové nástroje. Tento fakt může být dán tím, že vzrostl výpočetní výkon osobních počítačů a problémy spadající kdysi do kategorie super-počítání se pomalu dostávají do osobních počítačů blíže k běžnému uživateli. Příkladem může být zpracování videa, či tvorba realistických scén. Sice i dnes je třeba velký výkon, který není dostupný v kdejakém domácím počítači, ale například v rámci filmového průmysly a mezi firmami zabývajícími se filmovými efekty, je takto výkoná technika běžně dostupná. Za touto prací zcela určitě nestojí programátoři, ale umělci, lidé zabývající se animací, atp. Pro ty a ne jen pro ně, můžou být podobné nástroje velmi žádoucí a ani nemusí vědět, že právě „programují “.

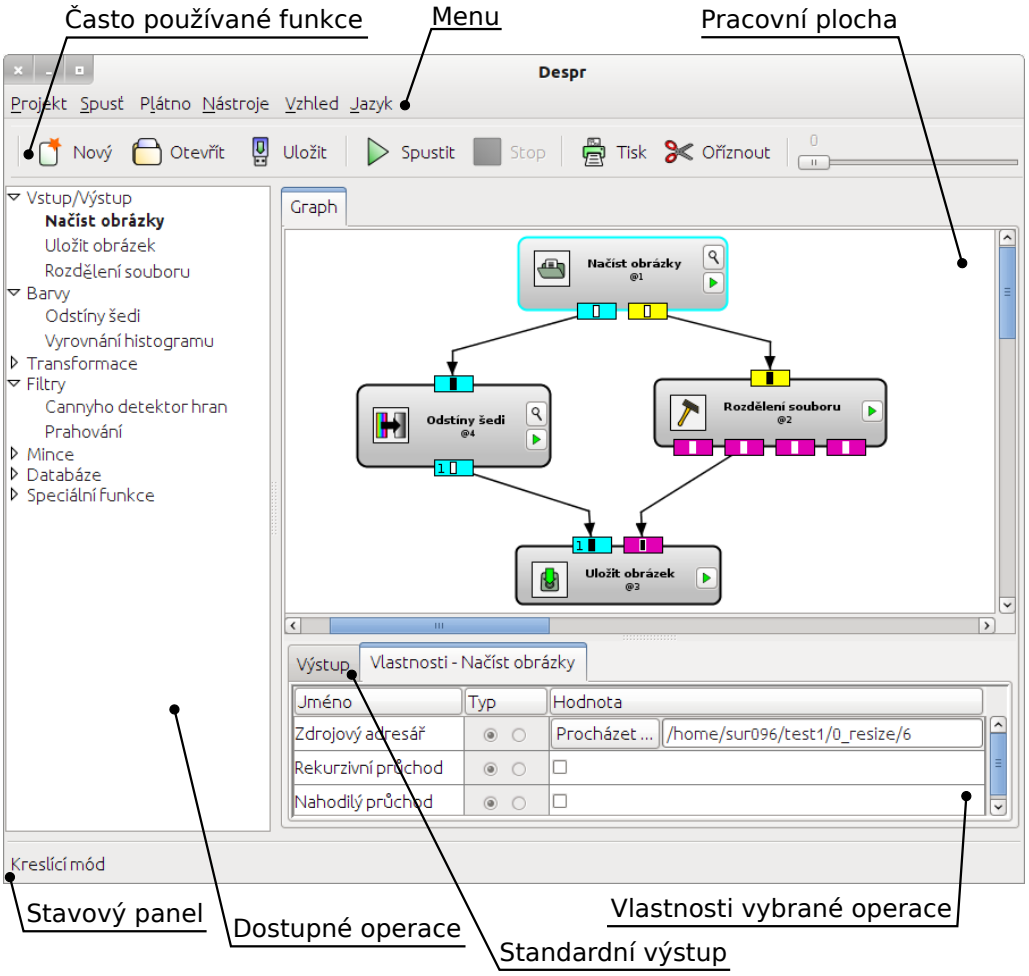
Nástroj vyvíjený v rámci této práce se pak pohybuje někde na pomezí mezi programátory a uživateli. Měl by být dostatečně jednoduchý na to, aby s ním mohl pracovat i běžný uživatel bez programátorských znalostí. Ale na druhou stranu i přívětivý pro programátory, aby pro ně nebylo těžké nástroj dále rozšiřovat.

Budoucnost této techniky je možná skryta v samotné definici na jejím počátku. Čímž je dataflow architektura a dataflow programovací jazyky. S tím rozdílem, že tyto „plovoucími data“ nebudou protékat mezi jednoduchými primitivními operacemi. Ale mezi komplexními funkcemi a stanou se z nich tak nástroje pro definici a správu komunikace. Což je možné vidět i dnes, např. MSVPL 3.3.3 umožňuje graficky naprogramovat i základní konstrukty, ovšem je to dobré, možná tak pro ukázku při výuce. Pro praktické řešení je to minimálně buď velmi neefektivní nebo zcela nepoužitelné. Proto je v prostředí robotics studia již předprogramována velká škála funkcí a metod pro práci s roboty. MSVPL tak spíše pomáhá jednoduše definovat toky dat mezi jednotlivými funkcemi.

# 4 Despr

Despr je název programu, který byl vyvinut v rámci této práce. Jak již bylo v předchozích částech naznačeno jedná se o nástroj pro návrh a spouštění složitějších algoritmů, poskládaných z jednodušších bloků. Tato část postupně představí celou aplikaci, od celkového pohledu, až po detailní popis jednotlivých komponent, z kterých se aplikace skládá.

Uživatelské rozhraní se víceméně drží původního návrhu, navíc byl pouze přidán panel, do kterého je přesměrován standardní výstup. To je z toho důvodu, že uživatelské operace mohou něco na standardní výstup vypisovat. Když se tak stane je vhodné, aby o tom uživatel věděl a nebylo to schováno někde v konzoly, která nemusí být ani spuštěná. Také byl přesunut panel s vlastnostmi jednotlivých operací pod pracovní plochu, protože šířka postranního panelu je pro něj malá a nevyhovující. Finální vzhled aplikace, s vyznačením hlavních komponent, je možné vidět na následujícím obrázku (obr. 18).

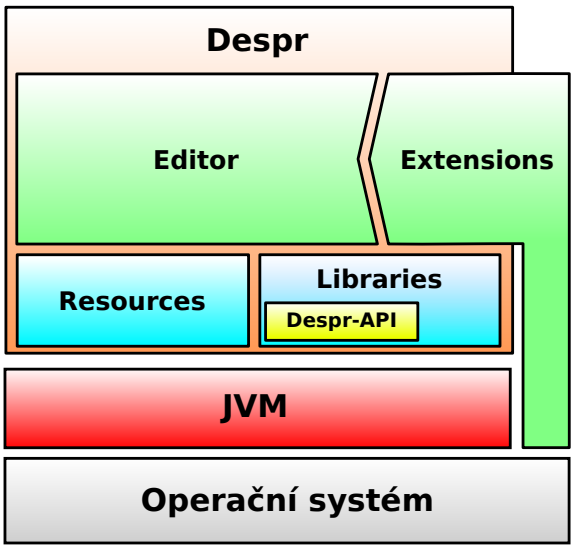



Obrázek 18: Despr - náhled, vč. vyznačení základních komponent.

V obrázku s náhledem aplikace (obr. 18) si je možné povšimnout, že i vizuální reprezentace operace se trochu liší od původního návrhu. Důvody těchto drobných úprav jsou podrobněji rozebrány v kapitole obecně se zabývající operacemi.

## 4.1 Celkový pohled

Aplikace byla navržena tak, aby bylo možné ji v budoucnu dále upravovat, s co nejmenší námahou. Pro tyto účely byl využit návrhový vzor MVC[11] [18]. Daný přistup je dodržen hlavně u komponent týkajících se grafu[12], ve kterém jsou modelovány a operací, kdy jsou od sebe odděleny, model, pohled (view) a ovládání (controller). U zbývajících částí je spíše využit systém *Model-Delegate* tak, jak je to známé z Javy a implementace Swingu [19]. Jedná se o systém, kdy ovládání a pohled jsou zkombinovány do jednoho objektu, do tzv. delegáta (delegate). Je to vhodné pro situace kdy ovládání pohledu je natolik specifické pro danou komponentu, že je zbytečné jej oddělovat a striktně se držet MVC. Ostatně v celé aplikaci je využíváno několik návrhových vzorů, avšak nikdy nejsou brány jako dogma. Je na ně pohlíženo, jako na doporučený postup a pro konkrétní situaci jsou buď modifikovány nebo různě kombinovány. Například pro definici grafu je využito MVC, ale určitá část ovládání, která je specifická pro manipulaci s pohledem, je do něj přesunuta. O tom ale podrobně v kapitole zabývající se implementací pracovní plochy.




Obrázek 19: Pohled na to z čeho se aplikace skládá a jak komunikuje s okolím.

Na obrázku 19 je vidět z čeho se aplikace skládá dohromady a jak komunikuje s okolím. Z obrázku je patrné, že jádro aplikace (blok *Despr*) jako takové je multiplatformní, resp.

[11] *Model-View-Controller*, jedná se o velmi známý přístup k tvorbě programů a v této práci není nijak dál teoretický nerozebírán.

[12] Pod pojmem graf se chápe model pracovní plochy.

funguje tam kde je k dispozici virtuální stroj javy[13]. Jádro je složeno z editoru, což je vlastně celá aplikace bez jakýkoliv rozšíření. Ty jsou uloženy externě (blok *Extensions*) a myslí se pod nimi všechny operace a rozšíření datových typů. Editor si je pak umí načíst a pracovat s nimi. Ovšem jak je vidět, rozšíření mohou využívat některých systémových prostředků, které jsou již specifické pro ten či onen operační systém nebo dokonce pro konkrétní instalaci, resp. to co je nainstalováno na daném počítači. V takové případě záleží čistě na programátorovi dané komponenty, jak moc ji vytvoří multiplatformní. Na druhou stranu tento požadavek nemusí být u rozšíření ani žádoucí a bylo by chybou uživateli explicitně zakázat využívat plně systémových prostředků. Příkladem takového užití mohou být funkce, které komunikují s databází. Ty prostě na počítači bez nainstalované databáze fungovat nebudou a pokud je uživatel chce využívat musí si databázi nainstalovat.

O úroveň níže jsou bloky *Resources* a *Libraries*. První blok patří čistě k editoru a reprezentuje zdroje nutné pro jeho spuštění. Těmito zdroji jsou ikony, konfigurační a lokalizační soubory. Druhý blok představuje knihovny, které mohou být sdílené nebo taky soukromé, tzn. patřící buď editoru nebo konkrétnímu rozšíření. Tak jako tak jsou všechny knihovny k aplikaci uloženy v jednom společném adresáři.

V bloku *Libraries* je vypíchnuta jedna důležitá knihovna - *Despr-API*. Jedná se o veřejné rozhraní nástroje, pomocí nějž je možné vytvářet vlastní operace a přidávat je do něj. Každé nové rozšíření, které tvoří ať již novou operaci nebo rozšíření datového typu musí implementovat určitá rozhraní z této knihovny. O tom která to jsou a jak se s nimi pracuje, ale blíže v části o možnostech rozšiřování.

Jak rozšíření, tak knihovny jsou „obyčejné“ jar balíky. Jediný rozdíl mezi nimi je, že knihovny jsou vyžadovány při kompilaci a rozšíření mohou být přidávána do editoru za běhu.

## 4.2 Veřejné rozhraní – API

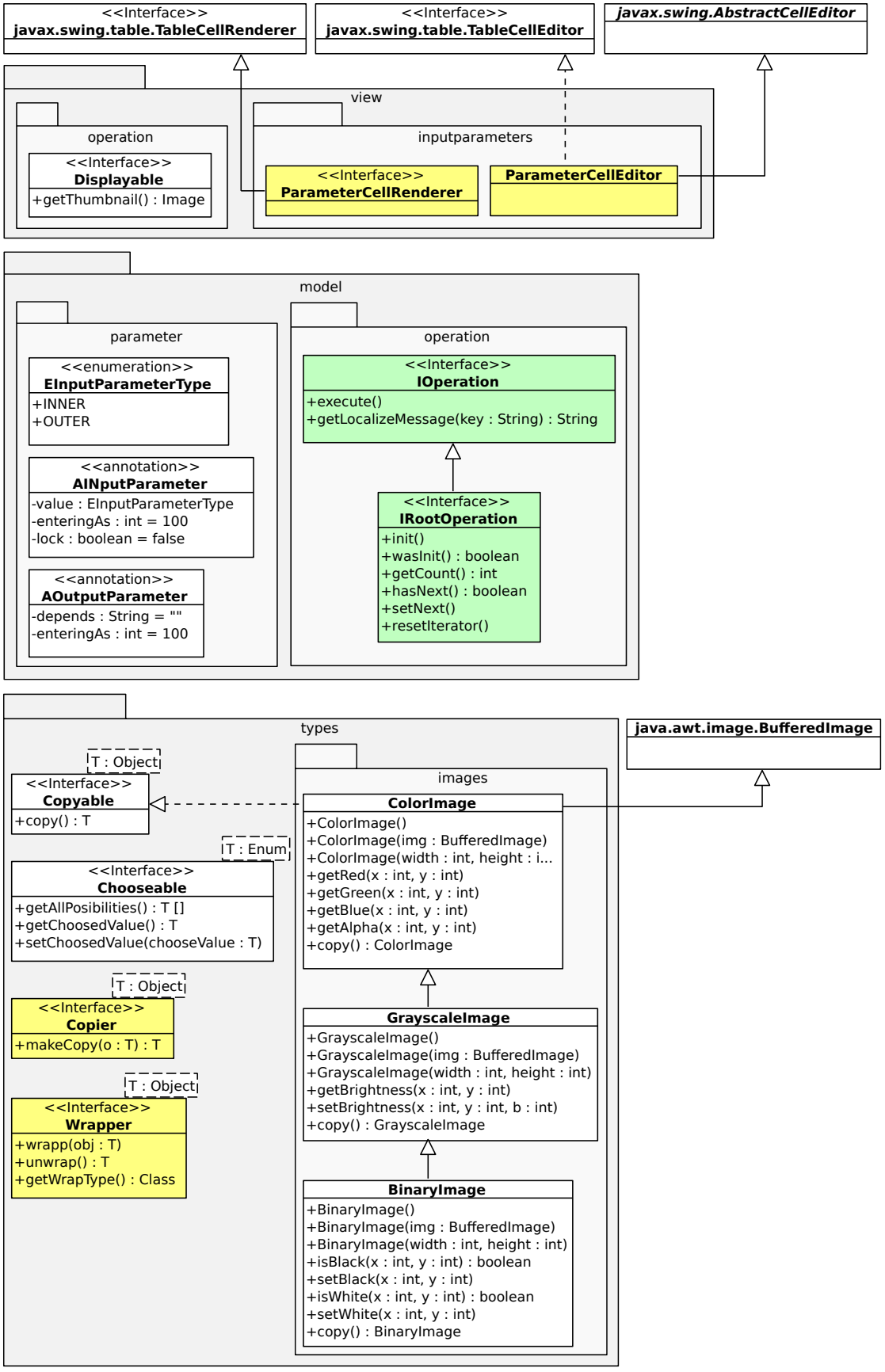
S aplikací je publikováno i její veřejné rozhraní,[14] které poskytuje vše nutné pro definici vlastích rozšíření. Jedná se o knihovnu definovaných tříd, rozhraní a anotací distribuovanou jako samostatný `jar` balík (`despr-api.jar`). Co všechno je součástí API ukazuje následující diagram, který je rozdělen do dvou obrázků (obr. 20 a 21). K rozhraní je dostupná i javadoc dokumentace, v které je možné nalézt, mimo jiné příklady implementace konkrétních rozšíření.

Na první pohled si je možné povšimnout, že některá rozhraní jsou barevně odlišena. Jedná se o ty, které přímo reprezentují možná rozšíření tak, jak byly zmiňovány již v předchozím textu. Zeleně označená rozhraní představují typy operací, na základě nichž může uživatel doprogramovat nové vlastní metody. Žlutě označená jsou pak ty, které tvoří rozšíření datových typů.

V první části diagramu (obr. 20) jsou balíčky `view`, `model` a `types`. Balík `view` obsahuje definice tříd a rozhraní ovlivňující vizuální část editoru. Pro operace je definováno rozhraní `Displayable`, které umožní zobrazit náhled výsledku dané operace.

[13] Při implementaci byla použita verze Java SE 6

[14] API je distribuované v archivu `despr-api_1.0.zip`. Kromě samotné knihovny jsou k dispozici i zdrojové kódy a javadoc dokumentace, adresáře `src` a `doc`.



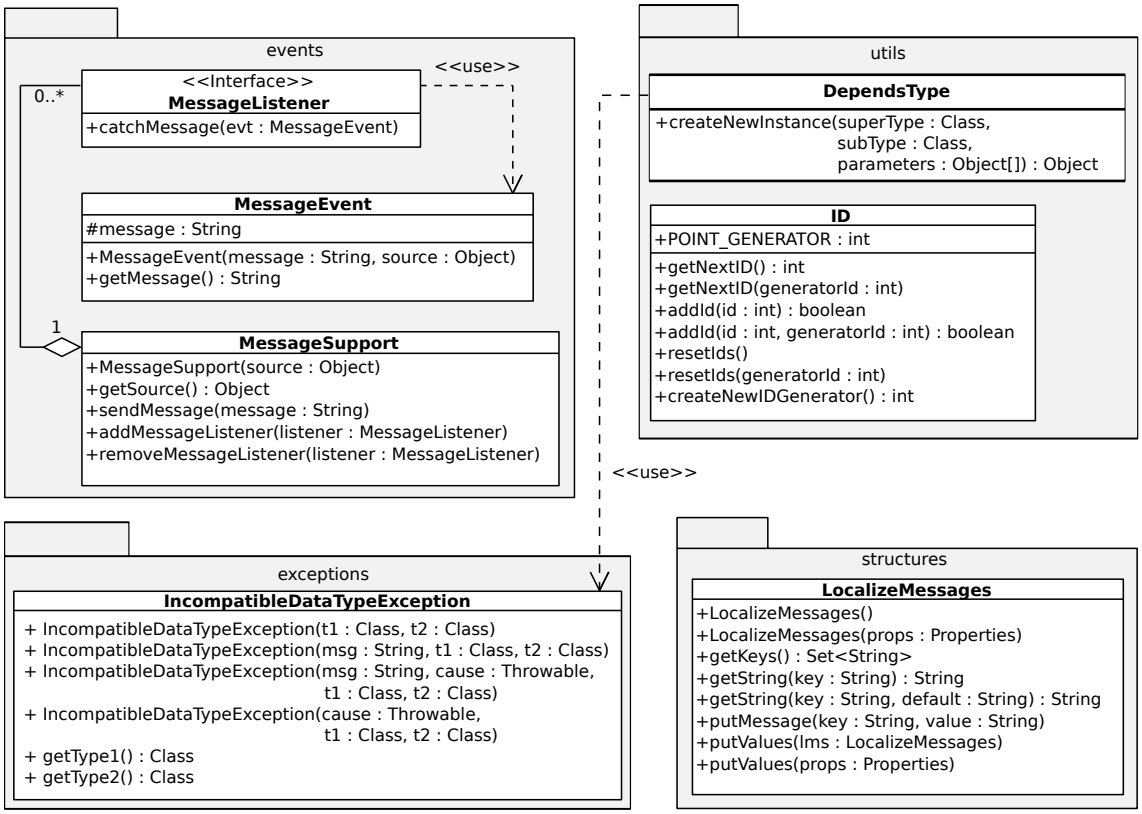

Obrázek 20: Struktura API $(^1/_2)$.

Pokud chce uživatel dát k dispozici tento náhled, musí být schopen daný výsledek nějakým způsobem graficky vizualizovat. Pro parametry operací, myšleno jejich datové typy (⇒ rozšíření datového typu), je možné definovat, tzv. editor (`ParameterCellEditor`) a renderer (`ParameterCellRenderer`). Oba dva slouží pro práci se **vstupními** parametry operace. První poskytuje komponentu pro editaci hodnoty parametru, druhý pak pro její vyobrazení. Nakonec je používá tabulka vlastností v editoru, kterou je možné vidět na obrázku 18 – *vlastnosti vybrané operace*.

Balíček `model` nabízí již zmiňovaná rozhraní pro definici vlastních operací (`IOperation` a `IRootOperation`). Pro parametry jsou zde k dispozici dvě anotace (`AInputParameter` a `AOutputParamtetr`), sloužící k oddělení vstupních a výstupních parametrů a výčtový (`EInputParameterType`), který navíc ještě rozděluje vstupní parametr na vnitřní a vnější. O tom jak tohle vše dohromady využít pro vytvoření vlastní operace, blíže v samostatné kapitole, která se věnuje rozšiřitelnosti nástroje.

Třetí balík `types` poskytuje další dvě rozhraní pro rozšíření datových typů a hlavně definuje typy obrázků. Nástroj je od počátku silně inspirován operacemi pro zpracování obrazu a typy, které java nabízí pro práci s obrazy nerozlišuje možné varianty. Proto jsou pro obrázky definované tři nové typy, na základě kterých je možné jednoduše rozlišit, zda se jedná o obrázek barevný (`ColorImage`), v odstínech šedi (`GrayscaleImage`) nebo černobílý (`BinaryImage`). Jednotlivé typy jsou hierarchicky seřazeny od obecnějšího po konkrétnější, což umožňuje v aplikaci pracovat například s černobílým obrázkem jako s barevným nebo s obrázkem v odstínech šedi, ale ne opačně. Toto nabízí větší variabilitu použití jednotlivých operací a navíc zavádí určitou syntaktickou kontrolu. Rozhraní `Copyable` a `Copier` se používají pro vytvoření kopie (nové instance) parametru se shodným nastavením. Tato vlastnost následně umožňuje výstupům operací, aby byly použity více než jedenkrát. Rozdíl je v tom, že první je možné využít při definici nového typu s tím, že je zároveň naprogramována metoda, která umí takovouto kopii vytvořit. Druhý je vhodný pro typy, které již existují, např. jakýkoliv javovský typ. Proto je toto rozhraní označeno jako rozšíření datového typu. Posledním takovýmto rozšířením je `Wrapper`. Ten je vhodný pro typy vstupních parametrů, jejichž hodnoty mají být uloženy a později znovu načteny. Despr umožňuje namodelovaný postup uložit a později i s nastavenými hodnotami načíst. K tomu je nutné, aby typy objektů reprezentující vstupní parametry, měli definovány přístupové metody (get/set) pro všechny své vnitřní položky, na základě jejichž hodnot je možné hodnotu daného parametru zrekonstruovat. Pokud typ takto definován není je možné pro to využít pravě `Wrapper`, který tyto nutné položky „vytáhne“, definuje k nim přístupové metody, plus metody pro zabalení a rozbalení objektu. Nakonec zbývá rozhraní `Chooseable` usnadňující práci s typy parametrů, které nabízejí výběr jedné položky z konečného počtu možných, typicky např. výčtový typ. Pro tyto typy je definován univerzální editor s rendererem a uživatel je tak nemusí pro každý nový výčet definovat zvlášť, pokud by je chce použít jako vstupní parametry, ale pouze stačí namapovat ty univerzální.

V druhé části diagramu, který pokračuje na obrázku 21 se nacházejí balíky s některými užitečnými funkcemi a strukturami. V prvním balíku `events` je k dispozici podpora pro




Obrázek 21: Struktura API $(^2/_2)$.

zasílaní textových zprav mezi různými objekty. Funguje podobně jako mechanismus výměny informací o změně vlastnosti v jave.[15]

Druhý balík `utils` obsahuje dva užitečné nástroje. První `DependsType`, poskytuje jednoduchou cestu jak vytvořit novou instanci závislého parametru. Jedná se o takový výstupní parametr, jehož datový typ závisí na datovém typu vstupního parametru téže operace. Jelikož aplikace umožňuje na konkretizované datové typy pohlížet jako na jejich obecnější definice,[16] pak pokud se této vlastnosti využije je vhodné obecný typ zkonkretizovat. Tato konkretizace se následně může projevit na některých výstupních parametrech operací. Například metoda pro změnu velikosti obrázku na vstupu přijímá obecně jakýkoliv barevný obrázek, ovšem pokud jí je předán obrázek černobílý pak je vhodné a žádoucí, aby i na výstupu bylo vidět že je černobílý obrázek, nikoliv barevný. Druhým užitečným nástrojem je generátor jednoznačných ID čísel, resp. seznam generátorů, kde každý zvlášť poskytuje jednoznačná ID, ne však mezi sebou. Tento generátor ve velké míře využívá editor, např. pro jednoznačné odlišení operací. Avšak v některých případech se hodí i v operacích a pro to byl zařazen do API.

[15] `PropertyChangeLisntener` a `PropertyChangeSupport`.

[16] S černobílým obrázkem je možné pracovat jako s barevným, atp.

Ve třetím balíku `exceptions` je definice výjimky zachycující případ, kdy je pokus spojit dohromady nekompatibilní datové typy. Výjimka je zde pouze z toho důvodu, že je použita v `DepensType`. Jinak ji využívá pouze editor a pro použití v operaci by byla vhodná pouze v případě, pokud by uživatel potřeboval řešit vytváření instancí, závislých parametrů, sám.

Poslední balík `structures` obsahuje strukturu sloužící k uchování lokalizačních zpráv. Ta je vhodná zejména v případech, kdy je třeba seznam lokalizačních zpráv aktualizovat, což neumožňuje klasický `ResourceBundle` definovaný v jave. Možnost aktualizovat seznam lokalizačních zpráv je vhodná zejména v případech, kdy lokalizační zprávy pochází z více lokalizačních souborů. Například lokalizační soubor nějaké specifikované třídy, obsahuje pouze zprávy pro nově přidané vlastnosti a zbylé zprávy načte z lokalizačního souboru rodičovské třídy.

## 4.3 Definice algoritmů – graf

Despr umožňuje vytvářet (sestavovat) nové algoritmy za použití jednodušších operací tím, že se vybrané operace natáhnou na pracovní plochu a definuje se, pomocí orientovaných hran, komunikace mezi nimi. Na tuto strukturu je možné pohlížet, jako na formu orientovaného grafu. Pro popis v textu práce i v rámci aplikace se tak používají, bez dalšího zavádění a vysvětlování, termíny z teorie grafů.

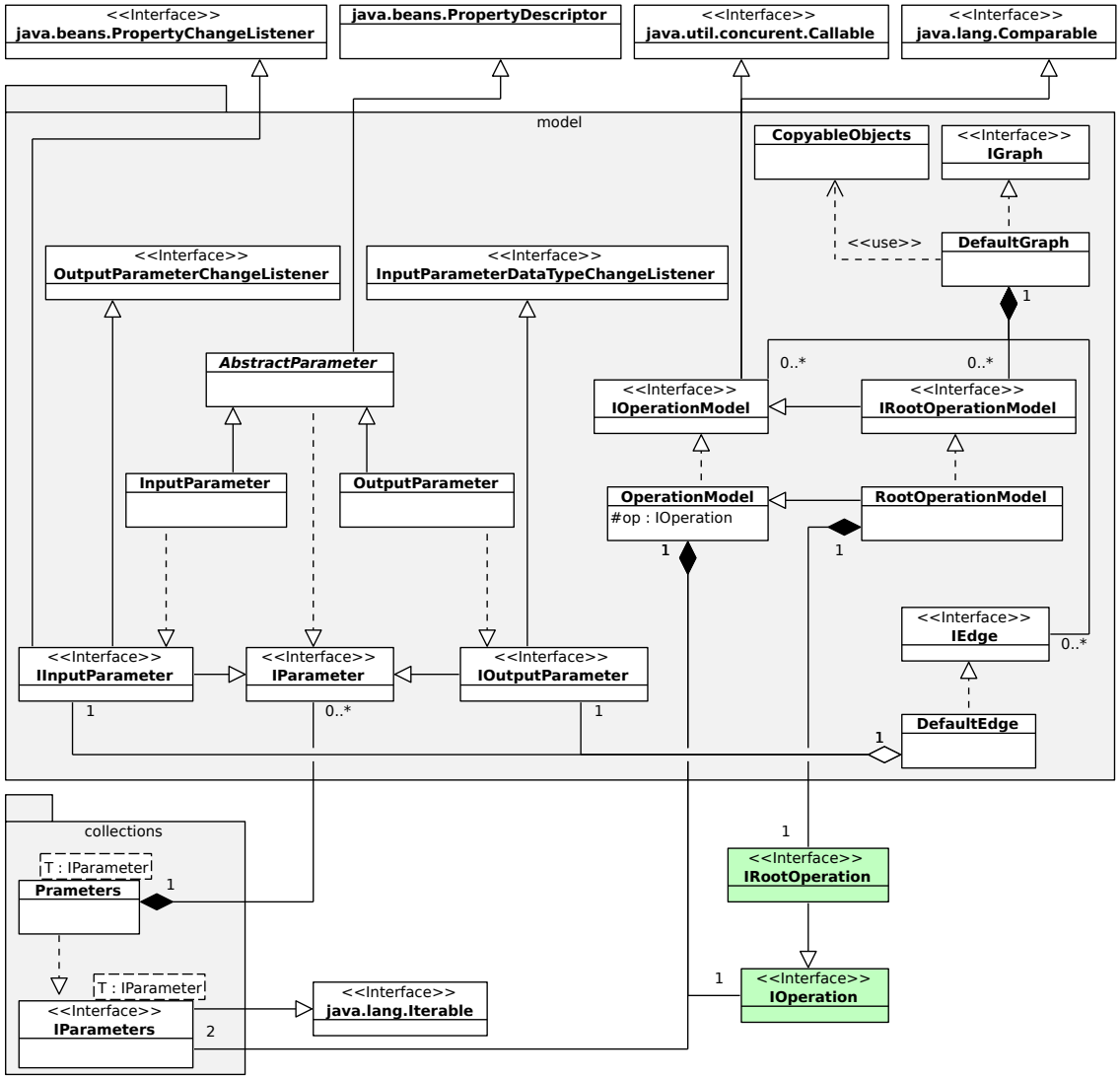
Ona pracovní plocha nebo též plátno grafu je vlastně vizuální reprezentace algoritmu, což je pouze jedna z částí. Celý koncept respektuje architekturu MVC. Je tak zvlášť definován model grafu, který postihuje jeho strukturu. Dále je odděleno ovládání (controller) starající se o vyhodnocení grafu, čímž je myšleno spuštění algoritmu. Na konec již zmiňované plátno grafu, které poskytuje grafický náhled a možnost jednoduché modifikace.

### 4.3.1 Model

V této kapitole je vysvětleno, jak je namodelovaný algoritmus (graf) reprezentován na pozadí aplikace, tedy to které třídy tvoří tzv. **model grafu** a jaké jsou vazby mezi nimi. Na obrázku 22 je třídní diagram popisující právě tyto vazby. V diagramu jsou vidět pouze jména tříd a rozhraní a to hlavní, jaké jsou vazby mezi nimi. Detaily o tom, co je v nich obsaženo, je možné vyčíst v javadoc dokumentaci, která je součástí příloh.

Velká část toho co tvoří model grafu se nachází v balíku `model`. Ten na nejvyšší úrovni reprezentuje rozhraní `IGraph`, které definuje metody pro práci s operacemi a hranami. Jeho implementaci představuje třída `DefaultGraph`. Ta v sobě uchovává dva seznamy operací, v prvním se nachází všechny použité operace `IOperationModel` a v druhém jsou zvlášť uloženy všechny kořenové operace IRootOperationModel pro snazší práci s nimi. Nakonec třída obsahuje seznam všech definovaných hran `IEdge`. Mimo balík si je možné povšimnout dvou rozhraní vyznačených zelenou barvou. Jedná se o rozhraní, která jsou obsažena v API a představují uživatelské operace. Zde je vidět jak tyto uživatelské operace zapadají do celého konceptu.

V levé polovině balíku se nachází třídy a rozhraní vztahující se k parametrům operací. Důležité je, že se dělí na vstupní (`IInputParameter`) a výstupní (`IOutputParameter`) parametry. Blíže o nich, ale až samostatné kapitole zabývající se parametry operací.

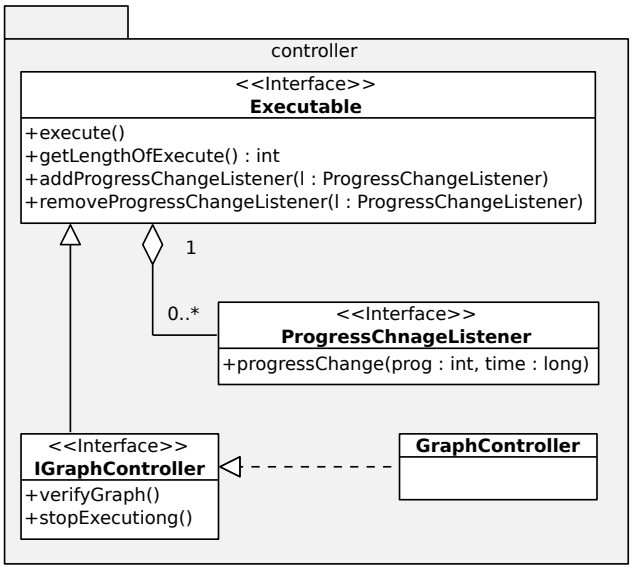



Obrázek 22: Struktura modelu.

### 4.3.2 Vyhodnocení grafu – controller

O zpracování grafu se stará třída `GraphController`, kterou je možné najít v balíku `controller`, diagram na obrázku 23. Pro třídu je definováno i rozhraní (`IGraphController`) pomocí nějž je například možné, relativně snadno, změnit způsob jakým je graf vyhodno-

cen.[17] V balíku se nachází ještě další dvě rozhraní, `Executable` a `ProgressChangeListener`. První představuje komponenty, které je možné spustit a sledovat to, jak daleko se výpočet nachází. Druhé rozhraní pak implementuje objekt, starající se o zobrazení stavu výpočtu. Aplikace tohoto mechanismu samozřejmě využívá a poskytuje informace o stavu zpracování, nejen grafu, ve stavové liště (obr. 18 – stavový panel). V této liště se vpravo při spuštění zpracování objeví progressbar. Rozhraní `Executable` se také využívá v akcích pro spuštění operace a načtení grafu ze souboru, kdy progressbar informuje o tom, že se něco děje.




Obrázek 23: Ovládaní grafu – struktura tříd.

Aby bylo možné graf vyhodnotit, je třeba aby splňoval určité náležitosti. Těmi jsou:

1. Graf nesmí obsahovat cykly,
2. všechny vstupní porty musí být využité (musí do nich vést hrana),
3. všechny vnitřní parametry musí mít nastavený korektní hodnotu,
4. musí existovat, alespoň jedna počáteční operace (operace bez vstupních portů),

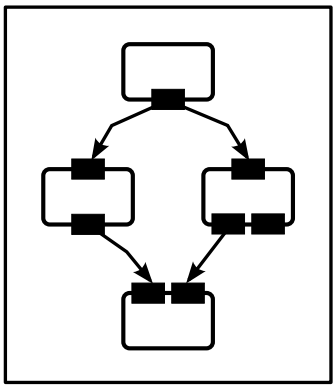
Graf by měl být acyklický hned z několika důvodů. Tím prvním, čistě praktickým, je snadné rozdělení operací do skupin, které je možné provádět paralelně. Druhým je, že to vylučuje uváznutí (dead-lock) aplikace, pokud tedy nedojde k uváznutí v rámci některé z operací. To by mohlo nastat pouze v případě, kdy by se výpočet v rámci operace také prováděl paralelně. Posledním je, že během vývoje a používání aplikace nebylo naraženo na jediný případ, kdy by se hodilo cykly zavést. Navíc nástroj dokonce

[17] Takováto změna se ovšem neobejde bez znovu zkompilování aplikace. Podpora přidávání nových grafů, jako je tomu u operací v nástroji není.

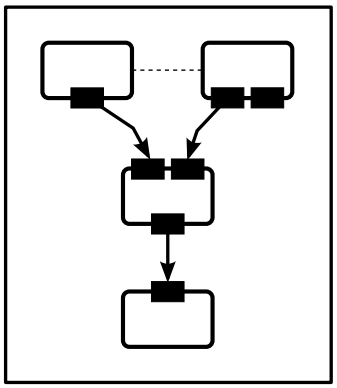
neumožňuje ani větvení, tj. podmínky. Je čistě navržen pro snadné definování komunikace mezi jednotlivými operacemi, ne na řízení běhu algoritmu.

Vstupní porty operací musí být využity, protože pokud by tomu tak nebylo, operaci by chyběla před spuštěním potřebná data. Ze stejného důvodu musí být také nastaveny všechny vnitřní parametry[18] operaci. Jelikož vstupní parametry se dělí na **vnitřní** (statické, nastavené uživatelem) a **vnější** (vstupní porty, do kterých posílají data jiné operace) je pak jasné, že operace potřebuje znát před spuštěním všechny vstupní hodnoty. Druhá a třetí podmínka tak víceméně vyjadřují to stejné, každá ovšem z trochu jiného pohledu.

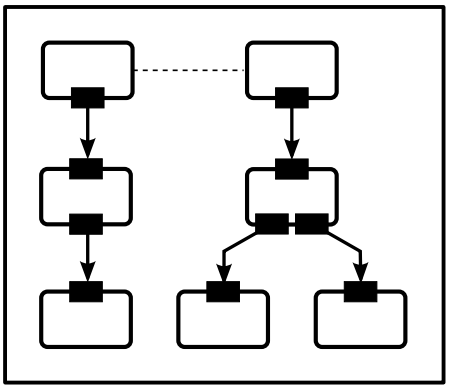
Poslední podmínka zdůrazňuje to, že namodelovaný postup musí někde začínat, tzn. musí existovat alespoň jedna operace bez vstupních portů. Jedná se o tzv. operace na nulté úrovni. Následující tři obrázky (obr. 24 až 26) ukazují příklady korektních grafů.



Obrázek 24: Jeden vstup, jeden výstup.



Obrázek 25: Dva vstupy.



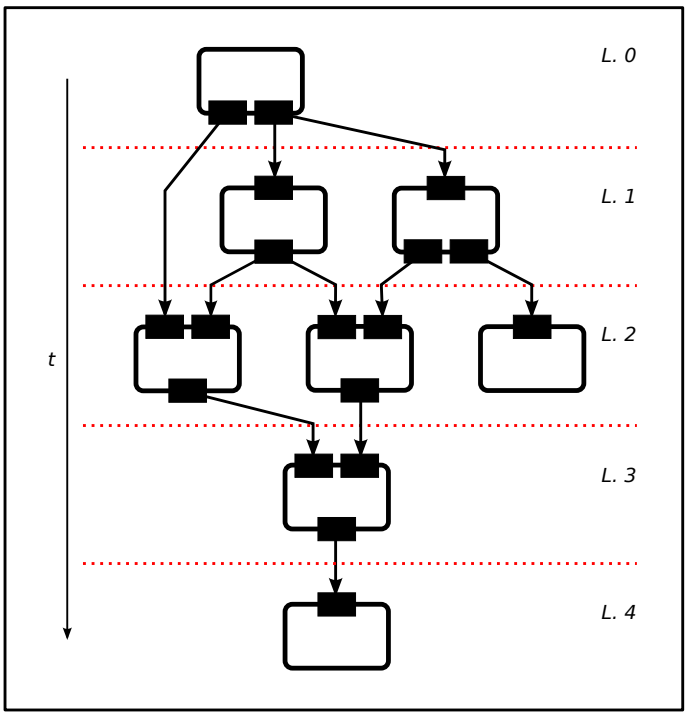
Obrázek 26: Dva „separátní“ grafy.

První příklad (obr. 24) znázorňuje takovou tu standardní situaci, kdy jedna počáteční operace poskytne data ke zpracování, následně probíhá samotný výpočet, který se může rozvětvit na více kroků a na konec se jednotlivé výsledky seskupí dohromady a uloží či jinak prezentují uživateli. Druhá situace (obr. 25) ukazuje případ, kdy je na vstupu více dat. To se hodí v situacích, jsou-li např. již nějaká data připravena předem. Algoritmus tak představuje, pokračování některého z předchozích výpočtů. Nebo jiná situace může být, kdy jedna z kořenových operací tvoří generátor dat (např. generátor jednoznačných ID). Z třetího případu (obr. 26) je vidět, že pokud má graf více kořenových operací, nemusí být nutně souvislý. Takových situací ovšem reálně moc nenastává, mohou sice vzniknout a pravidla to nezakazují, avšak i takovýto graf s dvěma komponentami ve skutečnosti úplně nesouvislý není. Operace na nulté úrovní se totiž nevyhodnocují zcela nezávisle. Výpočet je možné spustit pokud všechny počáteční operace poskytnou data ke zpracování, stačí aby to jedna z nich neudělala a výpočet končí. Tuto skrytou vazbu v obrázcích znázorňuje přerušovaná linka mezi počátečními operacemi. Vzhledem k tomu, že mohou existovat i kořenové operace, které nikdy neskončí (ony zmiňované generátory), pak minimálně jedna z těchto operací musí obsahovat konečný počet dat ke zpracování. Čímž je zajištěna

[18] Ty ovšem není možné zkontrolovat před spuštěním grafu. O jejich validaci se stará uživatelský kód a záleží na něm zda si data na vstupu validuje či nikoli.

konečnost výpočtu, prostě v určitou chvíli jedné z kořenových operací dojdou data a výpočet nemůže dále pokračovat.

Nyní již k tomu, jak probíhá samotné vyhodnocení grafu, resp. spuštění namodelovaného algoritmu, a které části se provádějí paralelně. Na následujícím obrázku (obr. 27) je příklad, jak by mohl takový namodelovaný postup vypadat. Před spuštěním je třeba, aby každá s operací „věděla“ na jaké úrovní (levelu) se nachází. Zařazení operací do jednotlivých úrovní se děje již, při vytváření grafu. Všechny kořenové operace a normální operace bez vstupních portů se nachází na úrovní nula (*L. 0*). Následně po každém přidaní hrany, daná operace zkontroluje zda má obsazeny všechny vstupní porty. Je-li tomu tak, vybere na základě všech předchozích operací, které jsou na ní napojeny, tu s nejvyšším číslem úrovně, přičte jedničku a zapamatuje si ji. Pokud jakýkoliv výstup z dané operace byl již připojen na jinou operaci, změna úrovně se propaguje dále tak, aby všechny operace měly stále aktuální informaci o tom kde se nachází. Před spuštěním je tak zajištěno, že všechny operace jsou zařazeny do správných úrovní.




Obrázek 27: Průběh zpracování, rozražení do jednotlivých úrovní.

Jak je z obrázku 27 vidět rozřazení do jednotlivých úrovní vytvoří seznam nezávislých operací, které je možno v jednom kroku spustit paralelně. Vyhodnocení pak probíhá tak, jak naznačuje časová osa. Seznam se projde od shora dolů a všechny operace nacházející se na dané úrovni se spustí najednou a čeká se dokud nejsou všechny dokončeny. Doba výpočtu jedné úrovně je tak určena nejpomalejší operací. Tímto způsobem je zajištěno, že jsou vždy spouštěny pouze operace, které mají všechna vstupní data a nejsou na sobě

nikterak závislé. To a nemožnost vytváření smyček zaručuje, že během průběhu zpracování nemůže dojít k uváznutí. Následující algoritmus (alg. 1) představuje pseudokód, který detailněji a hlavně přesněji popisuje onen průběh zpracování.

**Algoritmus 1** Průběh zpracování grafu.

```
1:  procedure EXECUTE(operations, root_operations, edges)
2:      if VEFIFY_GRAPH(operations, root_operations, edges) is OK then
3:          for all root_operations do
4:              root_op.iterator ← 0                ▷ Resetuje iterátor kořenové operaci.
5:          end for
6:          for all operations do                   ▷ Seskupení operací podle hodnoty: level.
7:              levels[op.level][count[op.level]] ← op
8:              count[op.level] ← count[op.level] + 1
9:          end for
10:         repeat
11:             for i ← 0, count_levels do
12:                 INVOKE_ALL(levels[i])           ▷ Paralelně spustí všechny operace na dané úrovni.
13:             end for
14:         until some root_op doesn't have next data ▷ Končí pokud jakákoli kořenová operace nemá
    další data.
15:     end if
16: end procedure
```

Jako úplně první krok se provede kontrola grafu, zda splňuje všechny náležitosti. O to se stará funkce VERIFY_GRAPH, jejíž pseudokód je možné vidět v dalším výpisu (alg. 2). Pokud je graf v pořádku nastaví se iterátory všem kořenovým operacím na počáteční hodnotu. V druhém kroku se rozřadí operace do skupin podle hodnoty úrovně, na které se nachází. Nyní je již vše připraveno pro samotné spuštění namodelovaného algoritmu. V cyklu se projdou všechny úrovně, kde operace na stejné úrovně se provádí paralelně. O paralelní spuštění operací se stará metoda INVOKE_ALL, která je součástí objekty typu `java.util.concurrent.ExecutorService`[19]. Proces zpracování se opakuje v cyklu, typu `do-while`, do té doby dokud mají všechny kořenové operace k dispozici data. Kontrola kořenových operací je až na konci cyklu z toho důvodu, že seznam těchto operací může být i prázdný. Avšak na nulté úrovní muže být jedna či více normálních operací, na jejichž základě se jedná o korektní graf, který je možné spustit.

Co všechno musí graf splňovat, aby bylo možné zahájit jeho zpracování je vidět v druhém pseudokódu (alg. 2). Jako první se zjistí zda existuje alespoň jedna kořenová operace s konečným počtem dat ke zpracování (řádek 4.). Každá taková operace si pamatuje počet dat, které má poskytnout. Pokud se jedná o generátor, který může poskytovat potenciálně nekonečné množství dat, pak má operace nastavenou hodnotu na: $-1$. Nyní je provedena kontrola (řádek. 9) zda graf obsahuje nějaké kořenové operace a pokud ano, musí alespoň jedna z nich obsahovat konečný počet dat. Je-li to v pořádku pak se dále

[19] Jako konkrétní implementace je použita třída: `java.util.concurrent.ScheduledThreadPoolExecutor`.

**Algoritmus 2** Kontrola grafu před spuštěním.

```
1:  function VERIFY_GRAPH(operations, root_operations, edges)
2:      exists_final_root_op ← false
3:      for all root_operations do
4:          if root_op.count_items > −1 then
5:              exists_final_root_op ← true
6:              break
7:          end if
8:      end for
9:      if exists_final_root_op = false ∧ root_operations isn't emty then
10:         return false ▷ Pokud existují nějaké kořenové operace a ani jedna z nich nemá konečný počet prvků
        ke zpracování.
11:     end if
12:     exists_zero_level_op ← false
13:     for all operations do
14:         if op isn't root operation ∧ op.level = 0 then
15:             exists_zero_level_op ← true
16:         end if
17:         for all op.input_parameters do          ▷ Kontrola zda jsou využity všechny vstupní porty.
18:             if parameter.type = OUTER ∧ parameter isn't used then
19:                 return false
20:             end if
21:         end for
22:     end for
23:     if root_operations is emty ∧ exists_zero_level_op = false then
24:         return false ▷ Pokud neexistuje žádná kořenová operace a zároveň neexistuje žádná jiná operace na
        nulté úrovní.
25:     end if
26:     for all edges do                            ▷ Ověření zda se v grafu nenachází nekorektní hana.
27:         if edge.incorrect = true then
28:             return false
29:         end if
30:     end for
31:     return true
32: end function
```

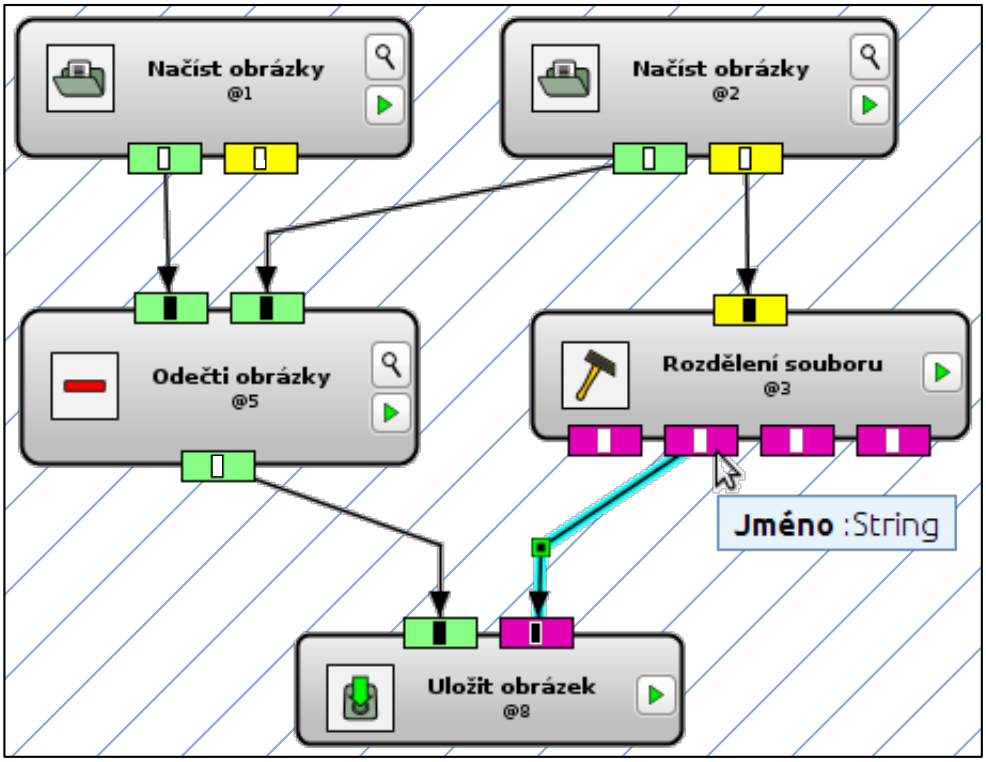
zjistí zda se na nulté úrovni nachází i nějaké normální operace. Pokud ano je možné graf prohlásit za korektní i kdyby neexistovala žádná kořenová operace (řádek 23). V rámci procházení seznamu operací se také kontroluje jsou-li všechny **vstupní** porty operací využity, tzn. vede do nich hrana (řádky 17 – 21). Jako poslední se ověří zda všechny hrany v grafu jsou korektní[20] (řádky 26 – 30). Dojde-li kontrola až na konec, je možné graf spustit.

[20]Nekorektní hrany mohou vzniknout mezi nekompatibilní typy.

### 4.3.3 Pohled

Interakci s uživatelem zajišťuje pracovní plocha nebo též plátno grafu. V kódu jej reprezentuje třída `GraphCanvas`, která se nachází v balíku `view`. Jedná se vlastně pouze o grafický kontejner,[21] který spravuje vizuální komponenty reprezentující operace a hrany. Podporuje takovou tu základní funkcionalitu, kterou by uživatel čekal od podobného programu. S prvky lze na plátně hýbat, mazat je, přidávat, atp.

Plátno uchovává jednotlivé grafické komponenty ve třech různých vrstvách. Na nejnižší vrstvě se nachází hrany. Nad nimi jsou záchytné body hran, pomocí kterých je možné měnit vzhled hrany. Na nejvyšší vrstvě, té nejblíže k uživateli, jsou samotné operace. Následující obrázek (obr. 28) ukazuje vizuální reprezentaci jednoho z příkladů[22], kde jsou vidět všechny zmiňované komponenty, které plátno spravuje. Plátno je pro názornost vyšrafováno a ohraničeno. Reálně se však jedná pouze o bílou plochu.



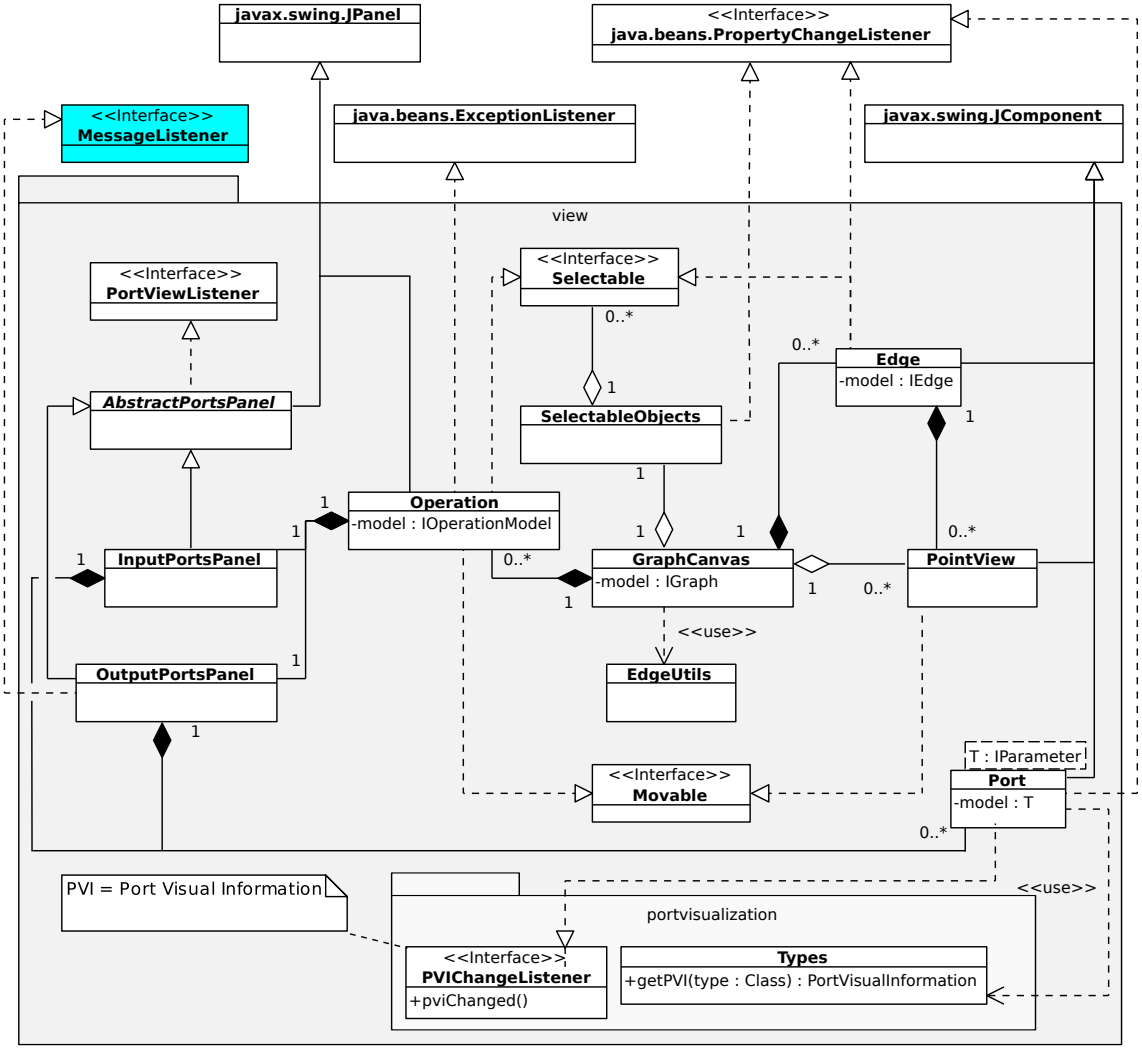

Obrázek 28: Ukázka grafu, porovnání dvou obrázků.

V této kapitole je ještě podrobněji popsáná struktura tříd, které implementují pohledovou část grafu. Třídní diagram je znázorněn na obrázku 29. Jsou v něm vyobrazeny pouze nezbytné třídy a rozhraní, vztahující se bezprostředně ke grafu tak, aby byla zachována srozumitelnost a čitelnost diagramu. Modře vyznačené rozhraní pochází z API.

[21] Podobně jako `JPanel`.

[22] Představuje algoritmus, který porovnává dva obrázky tak, že je od sebe odečte a výsledek zase uloží.

Balík ovšem ve skutečnosti obsahuje daleko větší množství tříd a rozhraní, mj. také implementace dalších grafických komponent použitých v aplikaci, jako jsou např. strom s operacemi nebo tabulka pro práci s vlastnostmi operací.[23] V diagramu také nejsou zahrnuty implementace společných funkcí, jako např. `MovableHandler`, který umožňuje pohybovat s komponentami typu `Movable`.




Obrázek 29: Pohled – struktura tříd.

Třídy `Operation` a `Edge` tvoří komponenty grafu a jak je v nich naznačeno přímo komunikují se svými protějšky v modelu. Plátno (`GraphCanvas`) si navíc uchovává odkazy na záchytné body hran (komponenty typu `PointView`). Jeden z nich je vidět na

[23] Tyto komponenty nebudou ani dále v textu podrobně rozebírány. Na jejich implementaci není vcelku nic až tak zajímavého. Více či méně podrobné popisy ke všem třídám je možné nalézt v javadoc dokumentaci nebo přímo v komentářích ve zdrojových kódech.

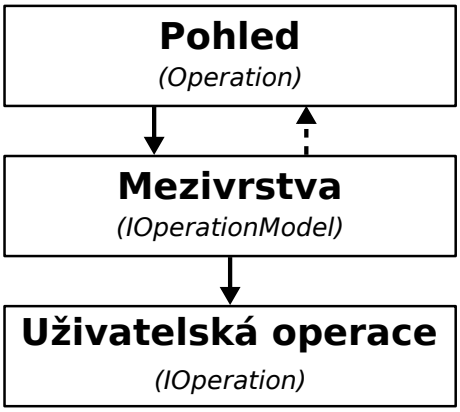
obrázku 28, černý čtvereček se zeleným orámováním. Záchytné body se zobrazují pouze, je-li hrana vybrána.

Dále jsou definována dvě důležitá rozhraní, `Selectable` a `Moveable`. První označuje komponenty, které je možné vybrat. Ty spravuje objekt typu `SelectedObjects`, který rozesílá informace o změně vybraných komponent registrovaným posluchačům[24]. Takovým posluchačem je např. tabulka vlastností, která poskytuje vstupní parametry právě vybrané operace. Druhé rozhraní tvoří objekty, s kterými je možné po plátně pohybovat. Jsou jimi operace a záchytné body hran. Je tak vidět, že s hranou jako takovou hýbat nelze. Měnit její polohu je možné pouze buď změnou pozice operace nebo záchytného bodu. Tím je zaručeno, že se hrana bude vždy nacházet mezi dvěma porty a nemůže tak „viset“ někde uprostřed plátna. Zbylé třídy vyznačené v diagramu budou podrobněji popsány v následujících kapitolách, které rozebírají jednotlivé komponenty grafu.

## 4.4 Operace

Operace tvoří základní stavební kameny namodelovaných algoritmů. Jedná se o uživatelsky definované funkce, které mohou být uvnitř různě složité. Na jednotlivé operace se pohlíží jako na atomické bloky a je k nim také tak přistupováno. Je-li operace naprogramována paralelně, její běh je považován za ukončený po dokončení metody `execute`. Zodpovědnost za korektní ukončení spuštěných procesů či vláken je na autorovi operace.

S operací se v rámci aplikace pracuje za pomocí několika komponent. Jedná se o tří vrstvou architekturu, kde vespod se nachází vlastní kód uživatelské operace, nahoře komponenta představující její grafickou reprezentaci v pohledu a mezi nimi část, která zprostředkovává komunikaci mezi pohledem a uživatelskou operací. Architekturu a komunikaci mezi jednotlivými vrstvami je vidět na obrázku 30. Plná šipka značí přímý přístup. Přerušovaná šípka označuje komunikaci řešenou podle návrhového vzoru pozorovatel (observer).




Obrázek 30: Jednotlivé vrstvy operace.

[24]Objekty typu `SelectObjectsChangeListener`. Rozhraní v obrázku není vyznačeno.

### 4.4.1 Uživatelská operace

Představuje kód, který musí uživatel napsat, pokud chce rozšířit nástroj o novou operaci. Jak již bylo v úvodu řečeno základním heslem, které se celou aplikací táhne jako červená nit' je **jednoduchost**. Vývojář, který chce přidat novou operaci do aplikace by tak neměl vynaložit větší úsilí, než je nezbytně nutné, pro její naprogramování. Z toho vyplývá, že kód se musí co nejvíce podobat kódu, který by musel být tak jako tak napsán.

Když je položená otázka: *Čím je každá operace (funkce) definována?* Odpověď je, že operaci tvoří vstupní, výstupní parametry a výkoná metoda, která na základě vstupních parametrů vypočte hodnoty těch výstupních. Více není třeba a po případném vývojáři také není o moc více vyžadováno. Ukázku jednoduché operace je možné vidět ve výpisu 1. Ukázková operace slouží jako generátor jedné náhodné celočíselné hodnoty.

```
public class RandomValue implements IOperation {

    private Random rnd;
    public RandomValue() {
        rnd = new Random();
        modulo = 10;
    }

    @AInputParameter(EInputParameterType.INNER)
    private Integer  modulo;

    @AOutputParameter
    private Integer  rndValue;

    public void execute() throws Exception {
        rndValue = rnd.nextInt ()  % modulo;
    }

    public String  getLocalizeMessage(String key) {
        return null;
    }

    public Integer  getModulo() {return modulo;}
    public void setModulo(Integer modulo) {this.modulo = modulo;}

    public Integer  getRndValue() {return rndValue;}
    public void setRndValue(Integer rndValue) {this.rndValue = rndValue;}
}
```

Výpis 1: Ukázka zdrojového kódu uživatelské operace; generátor náhodného čísla.

Kód operací se nechal inspirovat Java Bean konvencí a na první pohled jí i dost připomíná. Operace musí obsahovat bezparametrický konstruktor, který je možné použít pro nastavení výchozích hodnot proměnných. Ovšem hlavní důvod je ten, že operace jsou v aplikaci iniciovány podle jejich jména a je třeba, aby bylo možné vytvořit novou instanci, konkrétní operace, bez jakýkoliv dalších informací. Dále je nutné pro všechny vstupní a výstupní parametry implementovat přístupové metody – *get (is)* a *set*.

V rozhraní (`IOperation`), které definuje uživatelské operace se nachází dvě metody. Metoda `execute` představuje onu výkonou metodu, provádějící samotný výpočet. Druhá metoda `getLocalizeMessage` se v operacích nachází jelikož je celá aplikace lokalizována. A je vhodné, aby pro jména parametrů a operací existovaly tzv. „čitelné názvy“. Tím jsou myšleny takové názvy, aby v aplikaci např. parametr *rndValue* byl prezentován jako *Náhodná hodnota* a ne naopak. Ovšem pokud se vývojář nechce zabývat lokalizací není to nutně vyžadováno. Vrátí-li metoda hodnotu `null`, tak jako je tomu v ukázkovém příkladu, bude v aplikaci pro jména parametrů použito jejich pojmenování v kódu. Mělo by tak být minimálně dodrženo jakési „slušné“ pojmenování proměnných tak, aby se uživateli nezobrazil např. název parametru *prom1*, z kterého není možné určit o co se jedná.

Pro rozlišení vstupních a výstupních parametrů slouží anotace. Pro vstupní parametry se používá anotace `AInputParameter`. U ní je třeba, navíc ještě říci, zda se jedná o vnitřní (`INNER`) nebo vnější (`OUTER`) parametr. Vnitřní parametry jsou konstanty, jejichž hodnoty se nastaví na začátku výpočtu, před spuštěním. Naopak vnější parametry představují vstupy, které jsou zasílány výstupy jiných operací v průběhu výpočtu. V pohledu je představuje horní řada portů. Pro označení výstupních parametrů slouží druhá anotace, `AOutputParameter`. Na proměnné které nejsou označeny ani jednou z anotací se pohlíží, jako na soukromé či pomocné proměnné operace, jež využívá pouze ona sama a nejsou nikterak přístupné zvenčí. V příkladu je to proměnná `rnd`, z které jsou generovány náhodná čísla. Pro takovéto parametry není nutné definovat přístupové metody.

**4.4.1.1 Kořenové operace** Vzhledem k tomu že aplikace má také sloužit pro hromadné zpracování dat, byly definovány tzv. kořenové operace. Ty jsou reprezentovány rozhraním `IRootOperation`. To rozšiřuje původní rozhraní `IOperation` o metody, které umožní zpracovat celou kolekci dat. Fungují tak, že pro jeden průchod grafem vždy poskytnou jeden z prvků kolekce ke zpracování. Jakmile je běh na konci zjistí se, před úplným ukončením, zda kolekce kořenové operace obsahují další prvky ke zpracování, pokud ano běh se opakuje s novým prvkem. Lze si tak představit, že graf běží v cyklu a dokud všechny kořenové operace poskytují další data ke zpracovaní, jsou na ně aplikovány jednotlivé operace. Tímto způsobem se navržený algoritmus aplikuje na všechny prvky dané kolekce. Metody, které kořenové operace definují navíc jsou:

- `hasNext`, zjistí zda existuje další prvek ke zpracování.
- `setNext`, posune ukazatel na další prvek v kolekci.
- `resetIterator`, posune ukazatel na první prvek.
- `getCount`, zjistí kolik prvků je třeba zpracovat (velikost kolekce). Pokud metoda vrátí: $-1$, aplikace pohlíží na operaci jako na generátor, který je schopný poskytovat nové a nové data do nekonečna.
- `init`, připraví kolekci tak, aby možné volat čtyři předchozí metody. Například při zpracování obrazu načte cesty ke všem souborům, které se mají zpracovat a iniciuje iterátor.

- `wasInit`, zjistí zda byla operace inicializována. Kořenová operace může být během života konkrétní instance inicializována několikrát. To je dáno tím, že změna některých z jejich vstupních parametrů může zapříčinit změnu kolekce a je tak třeba ji znovu inicializovat.

```
public class RandomValues implements IRootOperation {

    private Random rnd;
    private boolean wasInit;
    private int idx;
    public RandomValues() {
        rnd = new Random();
        wasInit = false;
        modulo = size = 10; // default value
    }

@AInputParameter(value=EInputParameterType.INNER, lock=true)
private Integer size;
@AInputParameter(value=EInputParameterType.INNER, lock=true)
private Integer modulo;
@AOutputParameter
private Integer rndValue;

public void execute() throws Exception {
    if (!wasInit) init();
    rndValue = rnd.nextInt() % modulo;
}
public String getLocalizeMessage(String key) {return null;}

public boolean hasNext() {return idx < size;}
public void setNext() {idx++;}
public void resetIterator() {idx = 0;}
public int getCount() {return size;}
public void init() {
    resetIterator();
    wasInit = true;
}
public boolean wasInit() {return wasInit;}

public Integer getModulo() {return modulo;}
public void setModulo(Integer modulo) {this.modulo = modulo; wasInit = false;}
public Integer getSize() {return size;}
public void setSize(Integer size) {this.size = size; wasInit = false;}
public Integer getRndValue() {return rndValue;}
public void setRndValue(Integer rndValue) {this.rndValue = rndValue;}
}
```

Výpis 2: Ukázka zdrojového kódu kořenové operace.

V druhém příkladu (výpis. 2) je vidět kód jednoduché kořenové operace. Opět se jedná o generátor náhodných čísel, ale v tomto případě generuje kolekci náhodných čísel, nikoli jedno číslo. Namodelovaný postup tak bude aplikován na všechna vygenerovaná čísla.

Oproti předchozímu příkladu (výpis. 1) přibyl vstupní parametr `size` určující velikost kolekce. Navíc v metodách pro nastavení vstupních parametrů ovlivňující velikost kolekce a rozsah hodnot je řádek, který anuluje inicializaci (`wasInit = false`). To zapříčiní znovu-inicializování při jejich změně, čímž je vždy vygenerována taková kolekce, která splňuje vstupní podmínky. V tomto jednoduchém případě je sice pouze vynulován iterátor, ale ve složitějších případech, například operace pro načítaní souborů je třeba při změně adresáře znova načíst adresy souborů, zjistit kolik jich je a resetovat iterátor.

Nakonec si je možné povšimnout v anotaci vstupního parametru, že je použit „nový" argument `lock` a je nastaven na `true`. Anotace ve skutečnosti obsahují více parametrů ovlivňující práci a zobrazení parametrů, ovšem mají své výchozí hodnoty, které povětšinou dostačují. Jednotlivé argumenty anotací jsou detailněji popsány v kapitole zabývající se parametry. Použité nastavení značí to, že nelze přepínat typ vstupního parametru a je tak zafixován jeho počáteční typ, tedy jedná se pouze o vnitřní argument operace ($\Rightarrow$ nelze jej použít jako port). Takovéto chování je u kořenových operací doporučeno. Sice je možné mít i tento typ operace někde na nižší úrovni (tzn. má vstupní porty), ale pro tento účel nebyl nalezen vhodný příklad. Pokud se dále v praxi takové případy objeví, je možné použít kořenové operace i tímto způsobem. Ovšem tak jako tak, jsou všechny tyto operace vyhodnocovány společně. Tedy stále platí, že všechny musí mít k dispozici data, pro další běh algoritmu.

### 4.4.2 Mezivrstva

Mezivrstva zaobaluje uživatelské operace, obohacuje je o společnou funkcionalitu a zajišťuje jednotný přístup k jejím parametrům. Je navržena dle návrhového vzoru *služebník (servant)* [18]. Lze na ní pohlížet jako na rozšířený model operace, se kterým dále spolupracuje aplikace. Veškerý přístup k uživatelským operacím je řešen skrze ni, proto mezivrstva.

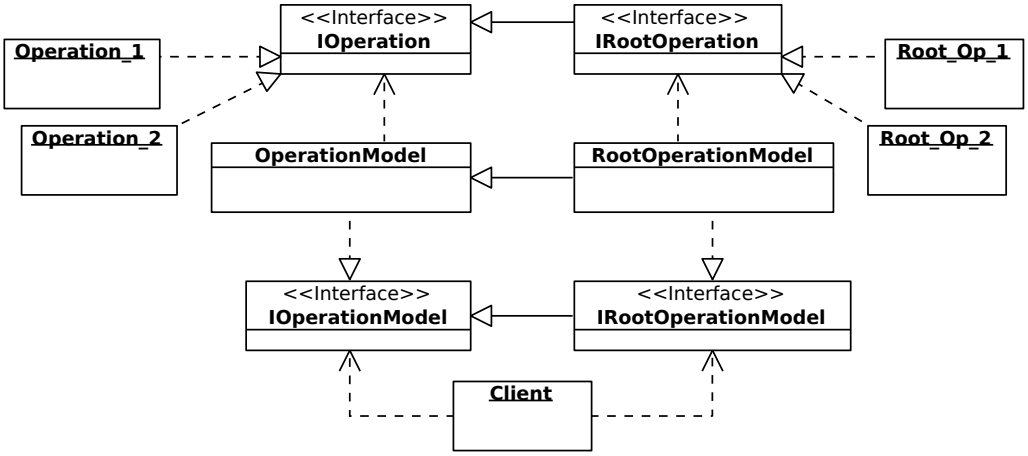
Implementace mezivrstvy obsahuje metodu, která pomocí reflexe „přečte" uživatelskou operaci. Ta jednotlivé parametry roztřídí do dvou seznamů, na vstupní a výstupní, podle použité anotace. Parametry které nemají anotaci jsou aplikaci skryté a využívá je pouze uživatelská operace. Jsou to jejich soukromé parametry. Pomocí seznamů se vstupními/výstupními parametry, je následně možné přistupovat k jakémukoliv parametru dané operace jednotným způsobem.

Pro každý parametr je vytvořena instance typu `IParameter`[25]. V ní jsou uloženy všechny důležité informace o parametru, jako jsou: jméno, přístupové metody, datový typ, hodnoty použité v anotaci, atd. Navíc jsou v mezivrstvě uchovány další pomocné informace k operaci, jako jsou: jednoznačné ID, číslo úrovně (levelu), na kterém se v rámci grafu nachází nebo též seznam zakázaných operací, kam nesmí daná operace posílat data[26].

Jak se mezivrstva používá je vidět na obrázku 31. Mezivrstvu reprezentuje rozhraní `IOperationModel`. Jeho implementace (`OperationModel`) pak používá objekty typu

[25] Respektive `IInputParameter/IOutputParameter`, podle anotace.

[26] Využívá se při detekci vzniku cyklu, kterému tak zabrání.



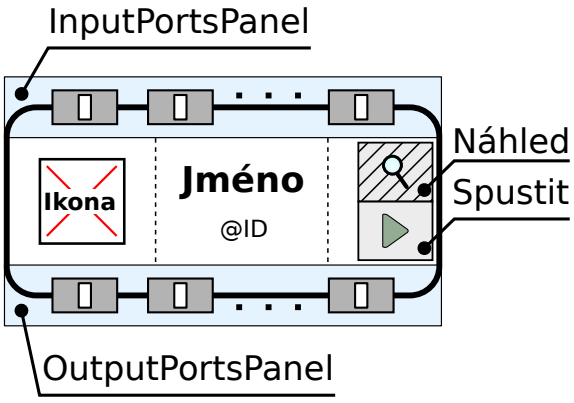

Obrázek 31: Použití mezivrstvy.

`IOperation`, které rozšiřuje a zprostředkovává komunikaci mezi uživatelskou operací a klientem (`Client`). Klient může být např. model grafu nebo grafická komponenta vizualizující operaci. Jelikož existují dva typy operací, je definována i mezivrstva pro kořenové operace. Tu reprezentuje rozhraní `IRootOperationModel`.

### 4.4.3 Pohled

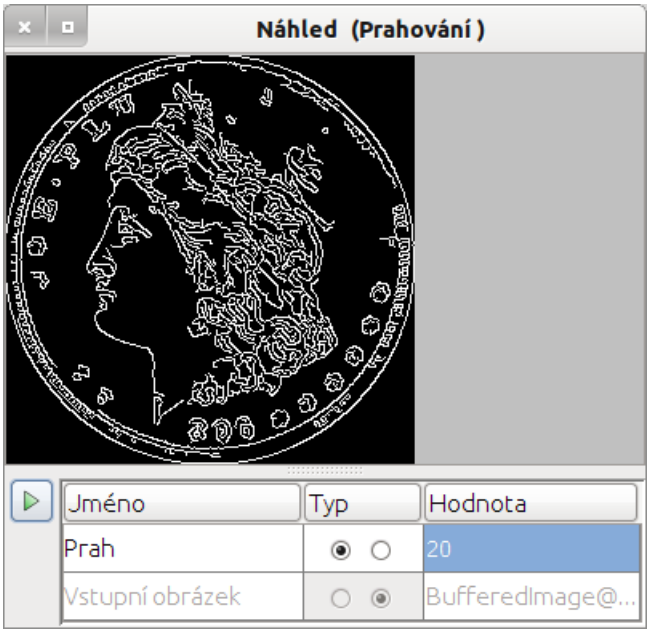
Grafická podoba operace se od návrhu (obr. 5) mírně liší. Finální rozvržení je vidět na obrázku 32. Hlavní změnou bylo, že ohraničení operace je navrženo tak, aby procházelo středem portů. Je to dáno tím, že na porty jsou napojovány hrany. Situace kdy se hrany napojovaly na ohraničení operace s tím, že porty byly uvnitř působila krkolomným dojmem. Proto se porty přesunuly jakoby na ohraničení operace, i když ve skutečnosti se jinak vykresluje ohraničení. Operace tak na plátně reálně zabírá více místa, než je na první pohled patrné. Kolik místa fakticky operace zabírají je naznačeno v ilustraci (obr. 32), kde modře vyznačené panely s porty mají nastaveno průhledné pozadí. Druhou změnou bylo, že přibyla tlačítka pro ovládání operace. Tlačítko *spustit* obsahuje každá s operací a umožňuje uživateli si projít graf (prokrokovat), operaci po operaci. Druhé tlačítko *náhled*, které se zobrazí pouze u operací implementujících rozhraní `Displayable`, umožní uživateli vidět výsledek operace. Okno s náhledem je možné vidět na obrázku 33.

Komponenta je generována na základě mezivrstvy. Na horní straně se nacházejí vstupní porty, na spodní pak výstupní. Vstupní porty jsou uloženy v `InputPortsPanel`, který komunikuje s panelem s výstupními porty, `OutputsPortPanel`. To že existují pro vstupní a výstupní porty dva různé panely si bylo možné všimnout již v diagramu postihující třídy starající se o pohledovou část grafu (obr. 29). Je to dáno tím, že je možné přepínat typy vstupních parametrů a ovlivňovat tak kolik vstupních portů a zda vůbec nějaké se na horní straně operace objeví. Tyto změny mohou měnit i rozměry komponenty a to zapříčiní, že se v rámci plátna změní i pozice výstupních portu, nikoliv však v rámci panelu, ve kterém se nachází. Panel s výstupními porty tak naslouchá tomu se vstupními a pokud




Obrázek 32: Grafická reprezentace operace.

nastane taková změna, která ovlivní rozměry operace, panel (`OutputPortsPanel`) informuje všechny své výstupní porty. Jsou-li na ně napojené nějaké hrany, jsou překresleny tak, aby byly stále připojeny k příslušným portům a nezačínaly někde kousek od nich.



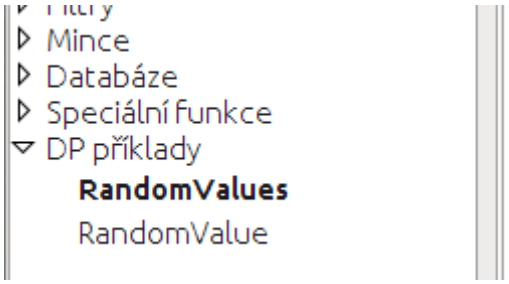

Obrázek 33: Okno s náhledem na výsledek operace.

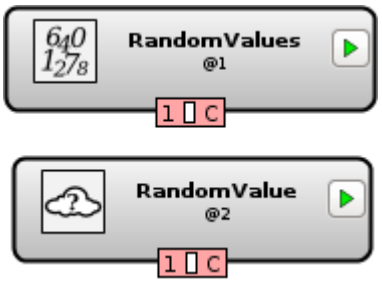
Jak již bylo řečeno operace implementující rozhraní `Displayable` mohou po zpracování i zobrazit grafickou reprezentaci výsledku. K tomu slouží okno s náhledem, které je možné vidět na obrázku 33. Okno nenabízí pouze náhled, ale také tabulku s vnitřními parametry operace, která se nachází vždy pod obrázkem. Vnitřní parametry je tak možné měnit přímo při zobrazeném náhledu a pomocí tlačítka *spustit*, vlevo od ní, se rovnou po-

dívat na výsledek. Okno s náhledem tedy hlavně slouží jako lepší možnost, jak nastavovat vnitřní parametry operací. Uživatel hned vidí, jak některé parametry ovlivňují výsledek a nemusí čekat na zpracování grafu, popř. si někde bokem ukládat mezi výsledky, aby věděl co se vlastně stalo.

Rozhraní `Displayable` definuje jedinou metody, která vrací výsledek ve formě obrázku. Mohlo by se tak na první pohled zdát, že slouží pouze pro zobrazování obrázku. Ve skutečnosti je možné takto prezentovat cokoliv co lze nakreslit, např. grafy. To že výsledkem funkce je obrázek je jen usnadňuje použití tohoto rozhraní. V aplikaci jsou, ale i případy, kdy funkce vrací vektor čísel, který je prezentován jako histogram. Možnostem použití se tak meze nekladou, je třeba je umět výsledek „nakreslit".

Na závěr této kapitoly je ještě ukázáno, jak vypadají vizualizované operace ze dvou předchozích příkladů (výpisy 1 a 2). V obrázku 34 jsou vidět operace zobrazené ve stromě operací. Je možné si povšimnout, že operace definované jako kořenové jsou ve stromě odlišeny pomocí tučného písma. Naopak na pracovní ploše je možné je od sebe rozlišit už jen pouze názvu, popřípadě ikony. Jak ve výsledku vypadají ony operace na pracovní ploše ukazuje obrázek 35. Tím, že ani jedna z operací neimplementuje rozhraní `Displayable`, mají na pravé straně pouze tlačítko, které spustí metodu `execute`.




Obrázek 34: Operace ve stromě operací. Obrázek 35: Operace na pracovní ploše.

## 4.5 Parametry

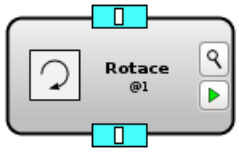
V předchozím textu se v několika souvislostech psalo o vstupních a výstupních parametrech operací. Tato kapitola podrobně rozebírá, jak se s parametry v aplikaci pracuje a jaká omezení jsou na ně kladena.

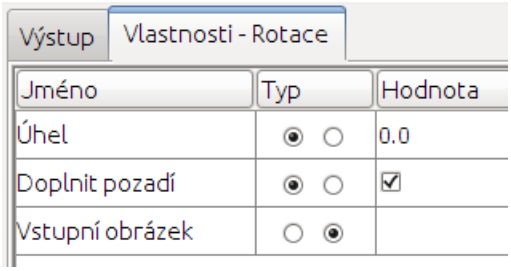
Prvně parametry se dělí na **vstupní** a **výstupní**. Vstupní parametry jsou v kódu označovány pomocí anotace `AInputParameter`. Pro výstupní parametry se používá druhé anotace `AOutputParameter`. Obě dvě je možné použít pouze nad proměnnými dané třídy, nikoli nad metodami, samotnými třídami a jinými konstrukty. Každá z anotací obsahuje několik parametrů, které ovlivňují další prací čí použití konkrétního vstupního/výstupního parametru. U obou anotací je definována vlastnost: `enteringAs`, jejíž defaultní hodnota je 100. Ta ovlivňuje řazení jednotlivých parametrů, ať již v rámci tabulky s nastavením vstupních parametrů operace nebo v seřazení portů na operaci.[27]

[27] Porty jsou řazeny klasicky zleva doprava.

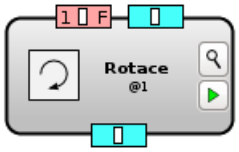
Hodnoty, které se zde zadávají, jsou celá kladná čísla, vč. nuly. Parametry, které používají výchozí hodnotu nebo mají nastavenu stejnou hodnotu, jsou dále řazeny lexikograficky podle jména. V první řadě je ale upřednostňováno nastavené pořadí.

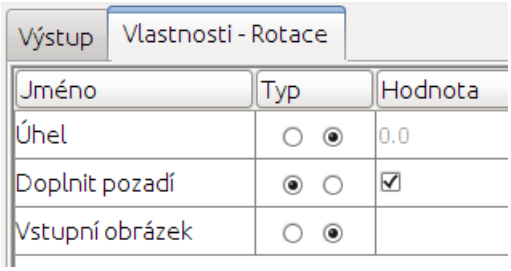
Anotace pro vstupní parametry obsahuje navíc další dvě vlastnosti, `value` a `lock`. První nemá žádnou výchozí hodnotu a je potřeba ji nastavit vždy. Hodnota je typu `EInputParameterType` a může nabývat dvou hodnot: `INNER` a `OUTER`. Rozřazuje tak vstupní parametry na vnitřní a vnější. Rozdíl mezi nimi je vidět na obrázcích 36 a 37.




Obrázek 36: Úhel jako vnitřní parametr.




Obrázek 37: Úhel jako vnější parametr.

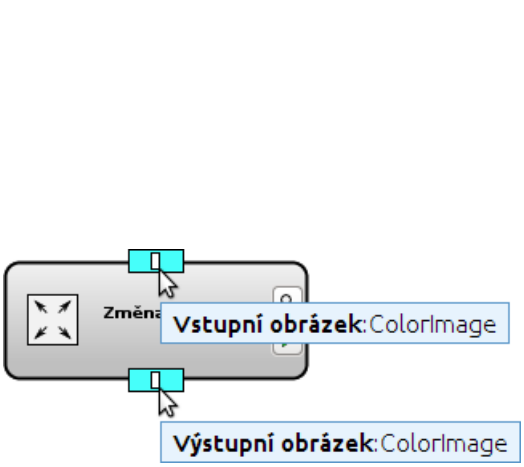
Pro přepínání typu vstupních parametrů slouží prostřední sloupec *typ* v tabulce vlastností operace. Přepínač má dvě možnosti korespondující s hodnotami výčtového typu `EInputParameterType`. Možnost vlevo reprezentuje vnitřní typ (`INNER`), možnost vpravo typ vnější (`OUTER`). Jak je vidět v obrázku 37 změna typu u parametru *úhel* z vnitřního na vnější zapříčiní, že možnost nastavení hodnoty v tabulce vlastností se uzamkne[28] a objeví se jako nový vstupní port operace. Přepínání typu je samozřejmě možné i naopak, s opačnými následky. Tato možnost přináší velmi velkou variabilitu použití jednotlivých operací. Stačí vytvořit novou operaci s nějakým defaultním nastavením[29] typu vstupních parametrů a dále již záleží na uživateli, jak vstupy použije. Motivací pro tuto funkcionalitu byla právě operace pro natočení obrazu. V kontextu této práce se totiž tato operace používá spíše ve druhém tvaru (obr. 37), jelikož natočení mincí na obrázku je třeba normovat a normující úhel je získán jako výsledek jiné operace[30]. Na druhou stranu očekávanější (pravděpodobnější) způsob použití, ze strany jiných uživatelů, je operace ve tvaru tak, jak je na obrázku 36. Pro to, aby nebylo nutné na pozadí (v modelu) definovat dvě operace natočení pro dva případy použití, byla zavedena možnost přepínat typ vstupních parametrů.

Poslední vlastnost `lock` má nastavenu výchozí hodnotu na `false` a slouží pro uzamčení přepínání typu vstupních parametrů. Pokud autor operace změní výchozí

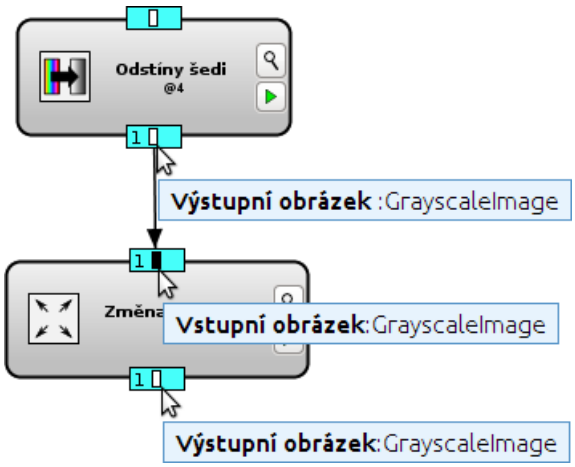
[28] Sloupec s hodnotou zešedne a hodnotu nelze editovat.
[29] Nejspíše podle odhadu nejčetnějšího použití konkrétní operace.
[30] Operace: *normování rotace*

hodnotu na `true`, pak ona možnost přepínáni vnitřních parametrů na vnější a naopak je potlačena. Parametr lze v takovém případě použít pouze v podobě, jaká byla defaultně nastavena. Tedy pokud autor rozhodl, že se jedná o vstupní port (vnější parametr) a možnost přepnutí uzamkne, není možné jej přepnout na vnitřní a nastavit konstantní hodnotu. Této možnosti se převážně využívá u kořenových operací, kdy jsou naopak uzamčeny vnitřní parametry tak, aby nebylo možné použit operaci nikde jinde než, na nulté úrovní v grafu.

Anotace pro vnější parametry (`AOutputParameter`) obsahuje navíc oproti společné vlastnosti, ještě vlastnost `depends`. Ta určuje *datově typovou závislost* na některém ze vstupních parametrů. Jako výchozí hodnota je použit prázdný řetězec, který značí, že datový typ výstupního parametru nezávisí na žádném ze vstupních parametrů. Nastaví-li autor operace nějakou závislost výstupnímu parametru, tzn. zadá jméno vstupního parametru, na kterém ten výstupní závisí, následkem je, že změna (konkretizace) datového typu vstupního parametru se promítne na specifikovaný výstupní parametr. Změna datového typu vstupního parametru je možná pouze tak, že se přenese tato informace jako výstup z jiné operace. Závislost lze tedy definovat mezi vstupními a výstupními porty. Této funkcionality využívá například operace pro změnu velikosti obrazu. Ta jako jeden ze vstupu přijímá barevný obrázek a na výstupu je taktéž barevný obrázek. Co se ale stane, když na vstup bude poslán černobílý obrázek? Pokud by nebyla definována závislost výstupního obrázku na vstupním, pak datový typ výstupního obrázku by zůstal jako barevný obrázek, což není to co se přirozeně očekává. Proto je definována závislost a nyní když přijde na vstup černobílý obrázek, objeví se černobílý obrázek i na výstupu. Jak vypadá použití závislých typů v aplikaci ukazují dva následující obrázky (obr. 38 a 39).




Obrázek 38: Výchozí stav vstupního a výstupního portu operace pro změnu velikosti.




Obrázek 39: Změna datového typu přicházející z operace převádějící obraz do stupňů šedi.

Na obrázku 38 je vidět výchozí stav portů operace, sloužící ke změně velikosti obrazu. Operace je schopná zmenšit, či zvětšit jakýkoliv typ obrázku a tím nejobecněj-

ším je barevný obrázek (`ColorImage`). Avšak jak je vidět v druhém příkladu (obr. 39) pokud je na vstup přiveden blíže specifikovaný typ, např. obrázek ve stupních šedi (`GrayscaleImage`), je změna typu propagována i na výstup, což přináší i určitou syntaktickou kontrolu. Ovšem samotné označení závislého parametru nestačí. Je třeba navíc zajistit korektní přetypování výstupního parametrů. K tomu je možné využít utilitu z API, která se jmenuje `DependsType` a její metodu `createNewInstance`. Vlastnost `depends` totiž pouze sváže výstupní typ se vstupním a zajistí předání informace o změně datového typu, které se projeví změnou vizuální reprezentace portu. Avšak o samotné vytvoření nové instance, korektního datového typu se musí postarat uživatel v metodě `execute`.

### 4.5.1 Datové typy

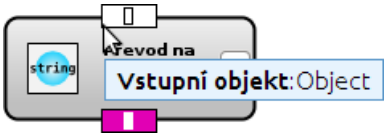
Na datové typy jednotlivých parametrů operací nejsou kladeny téměř žádná omezení. Jediné co by se jako omezení dalo nazvat, je zákaz použít primitivní datové typy pro parametry označené anotacemi `AInputParameter` a `AOutputParameter`. Což prakticky až tak omezení není, jelikož každý z primitivních typů má svou objektovou reprezentaci. V původním návrhu se tato možnost explicitně nezakazovala, ale v průběhu implementace to přinášelo mnohem víc problémů než užitku. V rámci jednotného přístupu k parametrům, tak byla možnost použití primitivních typů zakázána. Toto rozhodnutí navíc přineslo nemalé zjednodušení kódu a tím také redukci potenciálních chyb, které v něm mohly vznikat. Této vlastnosti si ji možné povšimnout i v ukázkách s kódy operací *RandomValue* a *RandomValues* (výpisy 1, 2), kde u soukromých (pomocných) proměnných operace, bez anotace, jsou použity klasické primitivní datové typy, avšak pro proměnné označené anotacemi jsou použity objektové verze datových typů.

Aplikace využívá typového systému javy, vč. výhod spojených s dědičností typů. Tento fakt byl nastíněn, již v minulém příkladě se závislými parametry. Ono totiž nelze přetypovávat parametry jen tak libovolně, ale musí splňovat podmínky javy[31] pro přetypování. To znamená, že konkretizovaný typ lze označit jeho obecnější formou, ale ne obráceně! To je důvod, proč je možné na vstupní port, který očekává barevný obrázek poslat obrázek černobílý. Operace, která umí pracovat s barevným obrázkem, pracuje stejně s obrázkem černobílým či v odstínech šedi. Naopak operace, která pracuje pouze s černobílým typem obrázků nemusí umět, a zpravidla ani neumí, pracovat s obrázky barevnými.

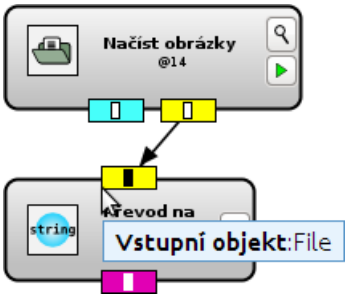
Pěkným příkladem teto vlastnosti je operace *převěd na řetězec* (*to string*). Jedná se o velice jednoduchou operaci, která na vstupu očekává libovolný objekt (typu `Object`) a jako výstup poskytne textový řetězec reprezentující daný objekt. Metoda `execute` této operace neudělá nic jiného, než že pouze zavolá metodu `toString` na objekt předaný na vstupu. Na takto definovaný vstup je možné poslat prakticky cokoli a metoda si s tím vždy poradí. Ukázka jak to vypadá v praxi je vidět na následujících obrázcích (obr. 40 až 42).

Na prvním obrázku (obr. 40) je operace bez přivedeného vstupu. Z tooltipu je vidět, že vstup je vskutku datového typu `Object`. Další dva obrázky (obr. 41 a 42) ukazují co se

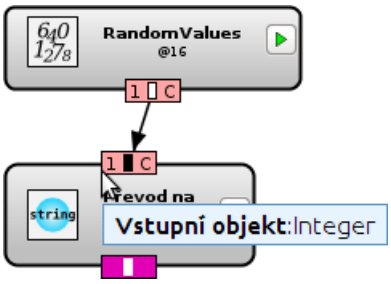
[31] Nejen javy, ale obecně podmínky objektových programovacích jazyků.




Obrázek 40: Operace bez přivedeného vstupu.




Obrázek 41: Operace přijímá vstup typu `File`.

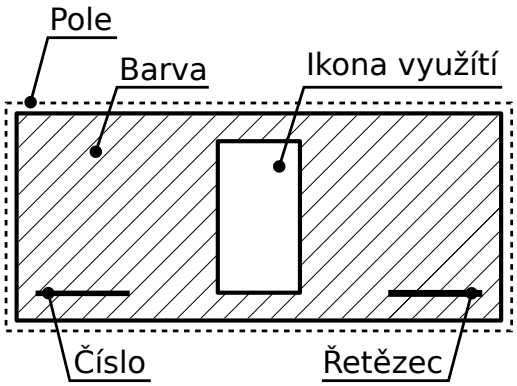



Obrázek 42: Operace přijímá vstup typu `Integer`.

stane po přivedení konkrétní hodnoty určitého datového typu. Nejen že dojde k onomu přetypovaní, jak naznačují tooltipy v obrázcích, ale také dojde k vizuální změně portu, která informuje, jaký typ je aktuálně na port přiveden.

#### 4.5.1.1 Porty

Port představuje vizuální prezentaci typu `IParameter`. Jedná se o společné rozhraní, které implementují jak vstupní, tak výstupní parametry. Rozhraní a třídy tvořící parametry jsou vidět na diagramu s modelem grafu (orb. 22). V několika předchozích ukázkách bylo vidět, že porty jsou různě barevné, objevují se na nich čísla, písmena a ve středu se nachází bílý či černý obdélníček. V této části je vysvětleno, co všechno daná symbolika značí, jak se čte a jak je možné toho využít.



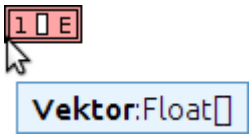

Obrázek 43: Vizualizace portu.

Na obrázku 43 je ilustrace portu, kde je popsáno, co vše může port obsahovat. Veškeré informace, až na *ikonu využití*, jsou uloženy v přepravce[32] `PortVisualInformation`, která se generuje ke každému datovému typu, resp. ke každému datovému typu použitého v rámci některého z portů, operace vložené na pracovní ploše. To znamená, že

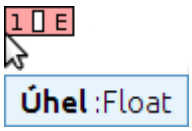
[32]Návrhový vzor přepravka z [18].

tyto informace se generují až když jsou potřeba. Co jednotlivé informace znamenají, je popsáno v následujícím výčtu.

**Pole** Pokud se jedná o pole určitého typu má zdvojené orámování. Zbývající informace se jinak vztahují k datovému typy polem. Je ovšem třeba dát pozor, protože nejsou rozlišeny různé dimenze polí, zdvojené orámování pouze upozorňuje na fakt, že se jedná o nějaké pole, nic víc o něm neříká. Rozdíl mezi portem reprezentujícím pole prvků a jeden prvek je vidět na obrázcích 44 a 45.

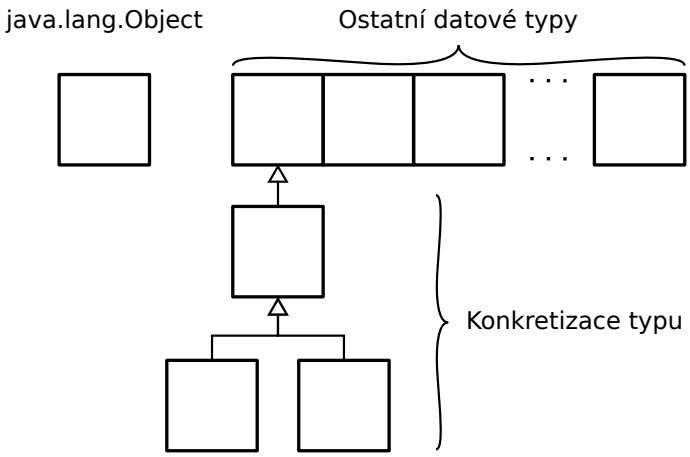



Obrázek 44: Port reprezentující pole prvků typu `Float`.




Obrázek 45: Port reprezentující jeden prvek typu `Float`.

**Barva** Barva od sebe odděluje, na první pohled, zcela nesouvisející typy. Pro každý datový typ, resp. pro jeho nejobecnější definici je vybrána barva, která by jej měla odlišit od ostatních. Jako na speciální typ se pohlíží na datový typ `Object`. Ten je uložen mimo strukturu typů a má přiřazeny bílou barvu. Je to dáno tím, že kdyby byl zařazen do struktury, jakýkoliv další datový typ zdědil by barvu předka. Tím pádem by měli všechny porty stejnou barvu.



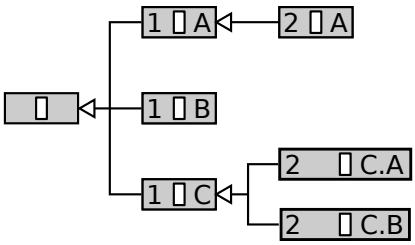

Obrázek 46: Přístup k datově typové struktuře.

Na obrázku 46 je vidět, jak se přistupuje k datovým typům. Je vytvořen seznam obecných datových typů *(ostatní datové typy)* s vyloučeným typem `Object`. Odlišná barva je přiřazena každému z typů v seznamu. Pokud se najde datový typ, který je konkretizací již některého typu v seznamu, je zařazen jako potomek do stromu

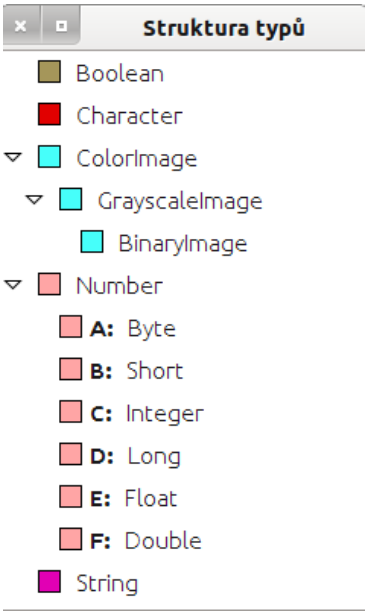
*(konkretizace typu)*. Seznam tak obsahuje odkazy na kořeny stromů, které představují nejobecnější definici datového typu. Podle barvy pozadí je také určena barva popředí, tak aby byla kontrastní. Proto se může stát, že některé porty mají text psaný bílou barvou a jiné zase černou. Toto nemá zvláštní význam, pouze zvyšuje čitelnost.

**Číslo** Číslo vyjadřuje míru specifikace typu nebo-li jinak hloubku v jaké se v daném stromě nachází. Typy s číslem nula (nejobecnější typy) jej nezobrazují. Uživatel tak může spojovat porty se stejnou barvou a navíc také může vést hrany z portů z vyšším číslem do portů s číslem nižším[33].

**Řetězec** Vzhledem k tomu, že vztahy mezi datovými typy mohou obecně vytvářet stromovou strukturu, je třeba od sebe rozlišit různé typy mající společného předka, avšak nacházející se na stejné úrovni. K tomu slouží specifikační řetězec. Ten tvoří písmeno či více písmen vedle sebe, pokud by bylo na jedné úrovní více typů než je písmen v abecedě. Má-li některý z typů předka, který byl označen specifikačním řetězcem, zdědí jej (podobně jako barvu). Pokud se následně nachází na stejné úrovní s více typy je za řetězec předka přidána tečka a nový specifikační řetězec.




Obrázek 47: Souvislost mezi vizuální reprezentací portu a jeho datovým typem.



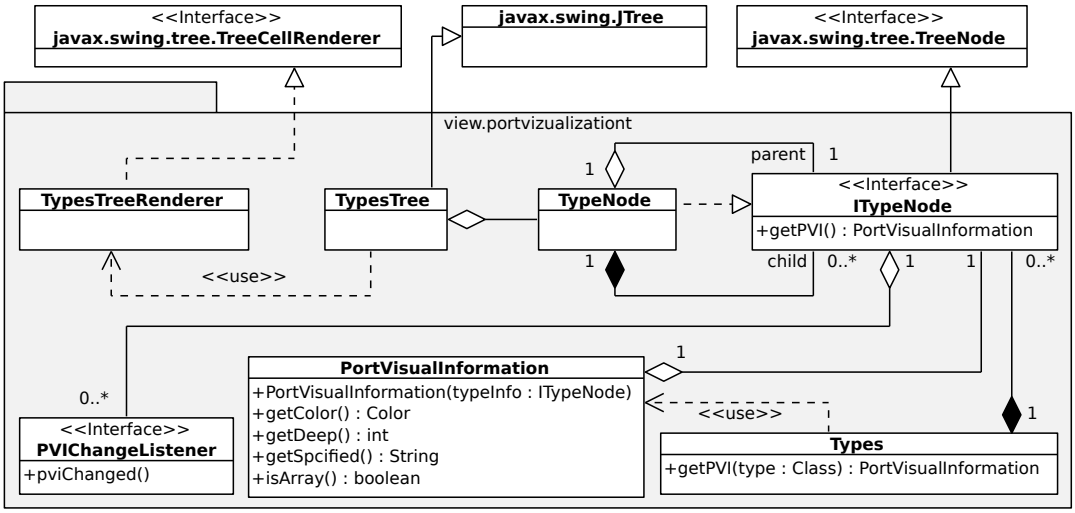

Obrázek 48: Okno se strukturou použitých typů

Na obrázku 47 je ukázka, jak by mohlo vypadat nějaké rozvětvení typů. Porty na nižší úrovni je pak možné spojovat s těmi na vyšší úrovni, podle směru šipky.

[33]Ne nutně o jedna nižším. Není třeba se vracet jakkoli v posloupnosti zpět nahoru. Prostě a jednoduše například z portu s červenou barvou a číslem deset může vést hrana do portu s červenou barvou bez čísla (s číslem nula) nebo do červeného portu s číslem šest, atp. Avšak ne naopak!

To znamená, že například z portu `(2 C.A)` je možné vést hranu, jak do stejně označeného portu, tak do portu označeného `(1 C)` nebo do kořenového portu bez označení. Druhý obrázek (obr. 48) je vidět okno se strukturou aktuálně použitých typů.[34] Uživatel má tak možnost podívat se jak aktuální struktura vypadá. Navíc je také možné v rámci tohoto okna změnit barvu vybranému typu, pokud by nebyla vyhovující.

Všechny výše zmiňované informace se generují automaticky. Třídy a rozhraní o to se starající se nachází v balíku `view.portvizualization`. Tento balík již bylo možné vidět v diagramu popisující pohled (obr. 29). Před tím, ale byly uvedeny pouze ty části, které komunikují s okolím. Kompletní strukturu balíku představuje následující diagram (obr. 49).




Obrázek 49: Struktura tříd a rozhraní starající se o vizualizaci portů.

Třída `Types` v sobě uchovává onu zmiňovanou strukturu na obrázku 46. Definuje veřejnou statickou metodu `getPVI(Class type)`, která poskytne informace nutné pro vizualizaci portu. V první řadě se pokusí daný typ nalézt, pokud se to nepodaří zařadí jej. Právě při tomto zařazování jsou generovány všechny potřebné informace. Z toho je vidět, že datově typová struktura se buduje dynamicky, v závislosti na datových typech, které se zrovna používají u portů. Aby po každém spuštění neměly porty různé barvy, je vygenerovaná barva spárována s konkrétním datovým typem a uložena do externího souboru[35]. Tím je zajištěna určitá konzistence tak, že když uživatel spustí aplikaci pokaždé uvidí „stejný graf“. Ovšem v rámci dvou různých instalací mohou být použité barvy různé. Vše záleží na tom, jak se barvy a typy na počátku spárují.

Jednotlivé stromy datových typů, resp. jejich uzly, reprezentuje rozhraní `ITypeNode` a jeho implementace `TypeNode`. Všechny informace k datovému typy, vč. informací nut-

[34] Nachází se v menu: `Nástroje → Struktura použitých typů`.

[35] `resources/SavedColors.properties`

ných pro vizualizaci portu jsou uloženy právě v něm. Portu je pak poskytnuta struktura `PortVisualInformation`, která mu zpřístupní pouze ty informace jež potřebuje vědět. Třída `TypesTree` představuje grafickou komponentu vizualizující aktuální typovou strukturu. Je použita v okně, které bylo vidět na obrázku 48.

Na ilustraci portu (obr. 43) byla ještě vyznačena *ikona využití*. Ta dává informaci o tom, je-li možné port použít, tzn. zda lze z něj nebo do něj vést hrana, resp. zda je schopný přijmout nebo poskytnout data. Ikonu tvoří ve výchozím stavu bílý obdélník s černým nebo bílým orámováním. Barva orámování je zvolena podle barvy pozadí, stejně jako u popisků uvnitř portu. Bílá ikona značí, že port je volný a k dipozici. Naopak má-li port černou ikonu použití, není schopen další data přijmout nebo v případě výstupního portu data poskytnout. Na obrázcích 50 a 51 jsou vidět příklady, jak vypadají nepoužité a použité porty. Jsou uvedeny obě varianty, jak s černým tak bílým orámováním.

Obrázek 50: Ukázka nepoužitých portů.

Obrázek 51: Ukázka použitých portů.

Jaká jsou pravidla pro určení toho, zda je port použitý či nikoli? U vstupních portů to je jednoduché, ty je možné použít pouze jednou. To znamená, že jakmile je do nich přivedena hrana, jeví se jako použité. Proto nelze do jednoho portu přivést více hran, ale vždy pouze jen jednu. U výstupních portů je to trochu jiné. Na první pohled by se mohlo zdát, že není důvod, proč by z výstupních portů nemohlo vést pokaždé libovolné množství hran. Problém je v tom, že jak vstupní tak výstupní parametry jsou reprezentovány objektem a jsou tak předávány odkazem na objekt, nikoli hodnotou. Proto nelze ze všech výstupních portů automaticky rozesílat data do různých vstupních portů. Může se tak stát, že u některých typů výstupních portů se po vytažení jedné hrany port uzamkne a již z něj nepůjdou získat data pro jiný vstup. Toto zabraňuje situacím, aby dvě nebo více operací pracovalo na jedněch datech a vzájemně si je tak měnily „pod rukama“.

Nicméně bez popisované funkcionality by aplikace téměř degradovala na to, že by jednotlivé operace bylo možné poskládat pouze sériově za sebe. Je tak třeba zajistit mechanismus, který bude umět vytvořit kopii (kopie)[36] dat na výstupním portu, jež bude možné přeposlat na jiný vstupní port. Pro tento učel jsou v API definována dvě rozhraní:

1. `Copyable`,
2. `Copier`.

Obě dvě mají stejný účel, avšak různá použití. První slouží pro nově definované uživatelské typy u nichž je možnost při vytváření implentovat dané rozhraní a automaticky tak bez dalších zásáhů z výstupních portů tohoto typu vést více hran. Příkladem mohou být typy rozlišující různé druhy obrázků (`ColorImage`, `GrayscaleImage` a `BinaryImage`), které jsou taktéž součástí API. Rozhraní `Copyable` definuje jedinou,

[36] Hlubokou kopii *(deep copy)*, tzn. dva odkazy na různé objekty se stejnými daty.

bezparametrickou metody `copy`, která by se měla postarat o vytvoření nové instance daného objektu. Příklad použití je v následujícím výpisu (výpis 3).

```
public class ColorImage extends BufferedImage implements Copyable<ColorImage> {

    ...

    @Override
    public ColorImage copy() {
        int width = getWidth();
        int height = getHeight();
        ColorImage copyImg = new ColorImage(width, height);

        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                copyImg.setRGB(x, y, getRGB(x, y));
            }
        }
        return copyImg;
    }
}
```

Výpis 3: Ukázka použití rozhraní `Copyable`.

Druhé rozhraní slouží jako rozšíření pro již existující datové typy. Jak java tak různé externí moduly definují mnoho různých typů a nebylo by zrovna vhodným řešením, aby autor komponenty pokaždé musel definovat `Copyable` verze typů, které chce využít. Proto existuje rozhraní `Copier`, pomocí něhož je možné ke konkrétnímu typu nezávisle definovat tzv. „kopírku“. V aplikaci je následně možné tento typ, s danou „kopírkou“ svázat, čímž je opět možné brát z takovéhoto typu výstupního portu data pro více vstupů.

Nevýhoda tohoto přístupu je, že je nutné typ a objekt umožnující vytvořit jeho kopii svázat dohromady. Není tak možné brát automaticky z daného portu více dat, jako je tomu u typu `Copyable`. Na druhou stranu ono spojení se provádí pouze jednou a všechny operace, které mají na vystupu porty typ pro který je již definována kopírující objekt, jej využívají automaticky. Výhoda je ta, že uživatel tak může používat standardní typy, která má k dispozici a pokud zjistí, že by potřeboval data daného typu rozesílat do více míst, pouze definuje objekt typu `Copier`, který bude umět kopii vytvořit nebo použije již hotový objekt, který by bylo možné na daný typ aplikovat. Pokud by takovou funkcionalitu ani nepotřeboval, nemusí vůbec objekt definovat.

Rozhraní `Copier` definuje, podobně jako to předchozí, jednu metodu, která umí objekt zkopírovat. Metoda `makeCopy(T object)` ovšem přebírá jako parametr objekt, jehož kopii má vytvořit. Tyto kopírující objekty jsou v aplikaci defaultně definovány pro všechny objektové verze primitivních typů, řetezce (`String`) a soubory (`File`). V následujícím příkladě (výpis 4) je vidět kód objektu `FileCopier`, který slouží jako kopírka pro soubory. Z něj je patrné, že většinu těchto objektů bude tvořit kód na pár řádků, pokud se tedy nebude jednat o nějaké příliš složité struktury, pro něž by bylo složité danou metodu definovat. V rámci celé aplikace byla „nejsložitější“ kopírovací metodou, metoda kopírující obrázek.

```
public class FileCopier implements Copier<File> {

    @Override
    public File  makeCopy(File t) {
        String  path = t.getPath();
        return new File(path);
    }
}
```

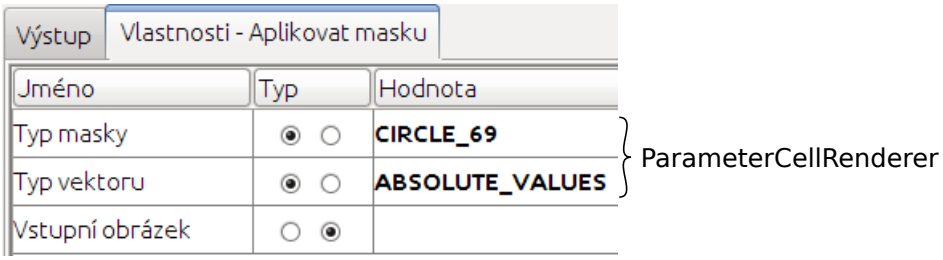
Výpis 4: Příklad použití rozhraní `Copier` pro kopírování objektů typu `File`.

Propojení typu s daným kopírujícím objektem, se provádí v aplikaci pomocí nástroje spravujícího rozíšření. O něm, ale až v samostatné kapitole, protože v následujícím textu budou ještě před tím popsána další tři typová rozšíření, podobnému rozhraní `Copier`, které je prvním z nich.

**4.5.1.2 Editace vstupních (vnitřních) parametrů** Pro editaci vstupních parametrů operace slouží tabulka *vlastnosti*. Tu bylo možné vidět již v několika ukázkách, např. v celkovém náhledu na aplikaci (obr. 18) nebo v příkladu s přepínáním vnitřních a vnějších vstupních parametrů (obr. 36 a 37). V něm byl také vysvětlen rozdíl mezi těmito dvěma typy vstupních parametrů. Tato část se zaměří pravě na vnitřní parametry a popíše co je třeba udělat, aby bylo možné editovat jejich hodnoty.

Vnitřní vstupní parametry jsou ty, které netvoří porty v horní části operace. Pro to, aby je bylo možné hodnoty nejen editovat, ale i rozumně prezentovat je třeba definovat další dvě typová rozšíření. První slouží pro zobrazení hodnoty a druhé pro její editaci.

Pro prezentaci hodnoty je v API definováno rozhraní `ParameterCellRenderer`, které není nic jiným, než přejmenováním rozhraní `TableCellRenderer`[37]. Je to kvůli typové kontrole při přidávání tohoto rozšíření tak, aby bylo jasné že má být použito v tabulce *vlastnosti* a nevztahuje se např. k nějaké z komponent, která nesouvisí se zobrazením vnitřního parametru. Příkladem takovéhoto rendereru může být `ChoosableCellRenderer` (výpis 5), který slouží pro prezentaci hodnot typu `Choosable`[38].




Obrázek 52: Ukázka operace obsahující hned dva vnitřní parametry typu `Choosable`.

[37] `javax.swing.table.TableCellRenderer`.

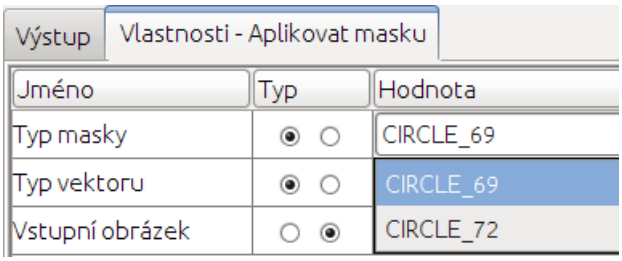
[38] Rozhraní `Choosable` je taktéž dostupné v API, a v kombinaci s výčtovým typem, je možné pomocí něj, celkem jednoduše, vytvořit seznam konstant, z kterého lze vybrat právě jednu.

Na obrázku 52 jsou vyznačeny dva parametry, pro které je použit zmiňovaný renderer. Jedná se o dva různé typy (`EMask` a `EVectorType`) implementující stejné rozhraní. Zde je vidět síla modulárního přístupu, stačí aby daná funkcionalita byla naprogramována jednou a dále je možné ji použít všude kde se hodí. Kód daného rendereru ukazující jeden z příkladů použití rozhraní `ParameterCellRenderer`, je vidět ve výpisu 5. Pokud k datovému typu není definován renderer, je použit defaultní, který pouze zavolá na daný objekt metodu `toString` a výsledek vypíše v tabulce.

```
public class ChoosableCellRenderer implements ParameterCellRenderer {

    private JLabel renderer;
    public ChoosableCellRenderer() {
        renderer = new JLabel();
        Font f  = renderer.getFont();
        renderer.setFont(f.deriveFont(Font.BOLD));
    }

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value,
            boolean isSelected, boolean hasFocus, int row, int column) {

        if  (value instanceof Choosable) {
            Choosable choosable = (Choosable) value;
            renderer.setText(choosable.getChoosedValue().name());
        }
        renderer.setBackground(table.getBackground());
        boolean isEditable = table.isEnabled() & table.getModel().isCellEditable(row, column);
        renderer.setEnabled(isEditable);
        return renderer;
    }
}
```

Výpis 5: Příklad použití rozhraní `ParameterCellRenderer`.

Pro editaci hodnot parametrů je definována abstraktní třída `ParameterCellEditor`. To rozšiřuje abstraktní třídu `AbstractCellEditor`[39], plus ještě implementuje rozhraní `TableCellEditor`[40]. Jedná se opět jako u rendereru o pojmenování typu, pro typovou




Obrázek 53: Použití editoru pro editaci typu `Choosable`.

[39] `javax.swing.AbstractCellEditor`.
[40] `javax.swing.table.CellEditor`.

kontrolu. Jako ukázka je použit `ChoosableCellEditor`, pro dokončení celého příkladu. Na obrázku 53 je vidět jak probíhá editace. Kód editoru je pak vidět v následujícím výpisu (výpis 6). V případě editoru je třeba myslet na to, že editor není pouhá komponenta, která prezentuje hodnotu, ale upravuje ji. Proto je pro ní nutné definovat akci, která obsluhuje změnu hodnoty. Na konci této akce by měla být zavolána metoda `fireEditingStopped` informující tabulku, že editace hodnoty byla dokončena. Jedná se o metodu, která je implementována v rámci abstraktní třídy `AbstracCellEditor`. Nakonec, pokud není pro daný datový typ definován editor, pak je editace hodnoty zakázána.

```
public class ChoosableCellEditor extends ParameterCellEditor {

    private Choosable choisedValue;
    private JComboBox editor;
    public ChoosableCellEditor() {
        editor  = new JComboBox();
        editor.addActionListener(new EditorAction());
    }

    @Override
    public Object getCellEditorValue()  {
        return choisedValue.getChoosedValue();
    }

    @Override
    public Component getTableCellEditorComponent(JTable table, Object value,
            boolean isSelected, int row,  int  column) {

        if  (value instanceof Choosable) {
            choisedValue = (Choosable) value;
            editor.setModel(new DefaultComboBoxModel(choisedValue.getAllPossibilities()));
            editor.setSelectedItem(choisedValue.getChoosedValue());
        }
        return editor;
    }

    class EditorAction implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            Object o = e.getSource();
            if  (o instanceof JComboBox) {
                JComboBox combobox = (JComboBox) o;
                Object selected = combobox.getSelectedItem();
                choisedValue.setChooseValue((Enum) selected);
                fireEditingStopped();
            }
        }
    }
}
```

Výpis 6: Příklad použití abstraktní třídy `ParameterCellEditor`.

## 4.6 Hrany

Hrany jsou v aplikaci reprezentovány, jak v modelu (`IEdge`) tak v pohledu (`Edge`). Avšak samotné propojení mezi porty je řešeno pomocí návrhového vzoru *pozorovatel (observer)* [18], kdy pokud jsou dva porty propojeny hranou, pak je model vstupního portu (`IInputParameter`) registrován u modelu výstupního portu (`IOutputParameter`) jako posluchač. Ten je reprezentován rozhraním `OutputParameterChangeListener`, které definuje tři metody:

1. `outputParameterValueChange`,
2. `outputParameterDataTypeChange`,
3. `outputParameterTabuListChange`.

Hrany tak slouží nejen k přenosu dat, ale i dalších informací.

První metoda slouží pro přenos dat z výstupního portu na asociované vstupní porty. Přenos se provádí vždy, po dokončení metody `execute` dané operace. Jelikož v tu dobu by měly všechny výstupní porty (parametry) již obsahovat data. O přeposílání dat se stará mezivrstva, autor operace musí pouze dodržet to, že v rámci metody `execute` budou naplněny daty všechny výstupní parametry.

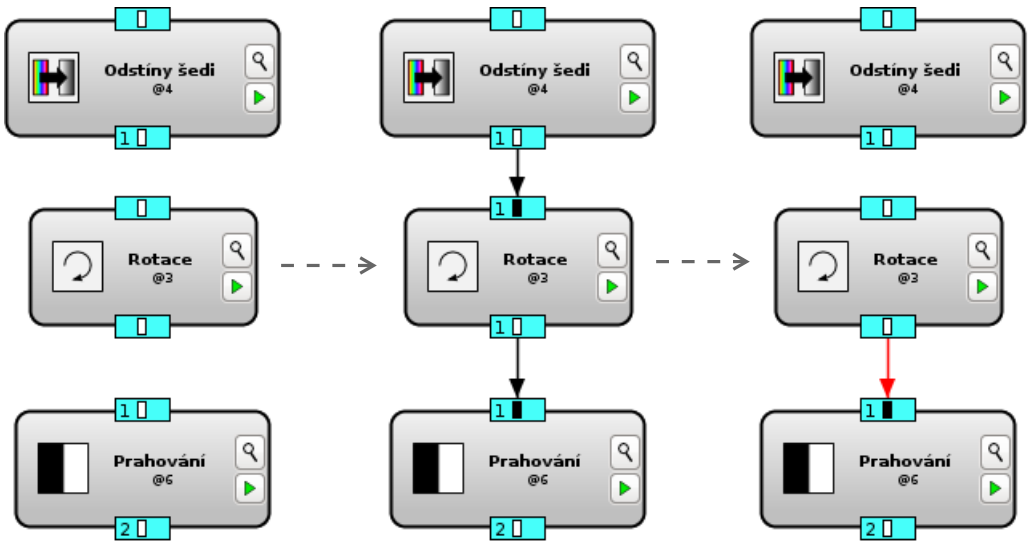
Druhá metoda rozesílá informaci o změně datového typu výstupního portu. Tato situace může nastat v případě závislých parametrů, kdy změna datového typu vstupního parametru zapříčiní změnu datového typu některého z výstupních parametrů. Závisle parametry byly detailněji popsány v kapitole 4.5. Metoda tedy pouze propaguje změnu do dalších operací, proto se v pohledu hrany nachází vždy mezi stejnými typy portů, i když původně měly parametry operací definovány datové typy jinak.

Poslední třetí metoda přeposílá seznam zakázaných operací. Ten je poslán vždy, když je hrana mezi dvěma porty přidána nebo smazána. V prvním případě si následující operace přidá seznam ke svému, v druhém jej odstraní. Seznam zakázaných operací zabraňuje vytvoření cyklu v grafu. Každá operace si tak musí pamatovat seznam operací, do kterých má zakázáno posílat data.

### 4.6.1 Model

Model hrany je tvořen rozhraním `IEdge` a jeho implementací `DefaultEdge`. Hrana obsahuje krom odkazu na vstupní a výstupní parametr, které spojuje také jednoznačné ID, pomocí nějž je možné identifikovat hranu mezi ostatními, plus informaci zda je či není hrana korektní.

Vzniku nekorektních hran je snaha předejít již, při jejich zadávání, např. hrany vytvářející cykly. Ovšem určitou posloupností kroků lze dosáhnout toho, že v grafu se objeví hrana spojující nekorektní typy. Jak k tomu může dojít je vidět na obrázku 54, kde v pravé části je jedna taková vyznačena, červenou barvou. Tato hrana nemůže vzniknout přímo, to je zakázáno a je vyvolána výjimka podobně jako v případě pokusu vytvořit cyklus. Ovšem díky závislým parametrům a možnosti přetypovávat vstupní parametry, zasláním




Obrázek 54: Posloupnost kroků vedoucí ke vzniku nekorektní hrany.

konkretizovaného typu do obecného, může vzniknout jedna čí více nekorektních hran při smazání jiné hrany.

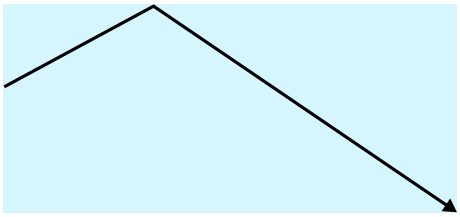
Obrázek 54 ukazuje posloupnost kroků, která vyústí vznikem nekorektní hrany. Za prvé je třeba vést hranu z konkretizovaného typu do nějaké operace s obecnou verzí daného datového typu, která změnu navíc promítne na jeden ze svých výstupních parametrů. Podobně jak ukazuje prostřední část v obrázku 54. To má za následek, že nyní je možné vést hranu z výstupního portu operace *rotace* do vstupního portu operace *prahování*, i když to před tím nebylo možné.[41] Avšak pokud je smazána hrana (*odstíny šedi* → *rotace*), která měla změnu na svědomí, jsou parametry přetypovány zpět do výchozího datového typu. Nekorektní hrany z grafu nejsou automaticky mazány. Jsou pouze vyznačeny a je ponecháno na uživateli, jak se s danou situací vypořádá. Tento stav může totiž nastat nechtěným smazání hrany, mající toto za následek a jejím vrácením se dá opět vše do pořádku. Graf obsahující nekorektní hrany, nelze samozřejmě spustit, protože neprojde kontrolou.

### 4.6.2 Pohled

Pohled tvoří třída `Edge` představující grafickou komponentu. Hrana je reprezentována lomenou čarou zakončenou šípkou zdůrazňující směr. Co je na pohledu zajímavé, je že tvoří klasickou swingovskou komponentu[42], což přináší v případě hrany určité problémy. Následující ilustrace (obr. 55) ukazuje, jak každá hrana vypadá ve skutečnosti.

[41] Toto je hlavní důvod proč si parametry pamatují datový typ aktuální hodnoty a ne pouze ten obecně definovaný při vytváření operace.

[42] Dědí z třídy `JComponent`



Obrázek 55: Ilustrace komponenty představující hranu.

Jak je vidět hranu tvoří obdélníková komponenta v rámci níž je hrana vykreslena. Hlavním problémem je, že k takovéto komponentě nelze přidat přímo reakce na akce myši, protože ta reaguje na celou oblast komponenty, vč. vyznačeného pozadí. To je ovšem nežádoucí, jelikož na myš by měla reagovat pouze vykreslená hrana. Dalším důvodem je, že se jednotlivé hrany mohou různě překrývat a hrana, která by byla někde úplně vespod, by na myš nereagovala vůbec. Reakce myši, na které zdánlivě reagují hrany, jsou ve skutečnosti registrovány u pracovního plátna `GraphCanvas`. To využívá třídy `EdgeUtils`[43] poskytující metody, které umí zjistit zda se kurzor myši nachází na hraně, popř. i danou hranu vybrat. Výběr hrany se provádí na základě výpočtu vzdálenosti kurzoru od všech hran, hrana která je nejblíže kurzoru je vybrána. Tato oklika řeší to, že „hrany reagují" na události myši tak, jak uživatel přirozeně očekává, i když ve skutečnosti se o jejich obsluhu stará pracovní plátno.

## 4.7 Ukládání, načítaní grafu

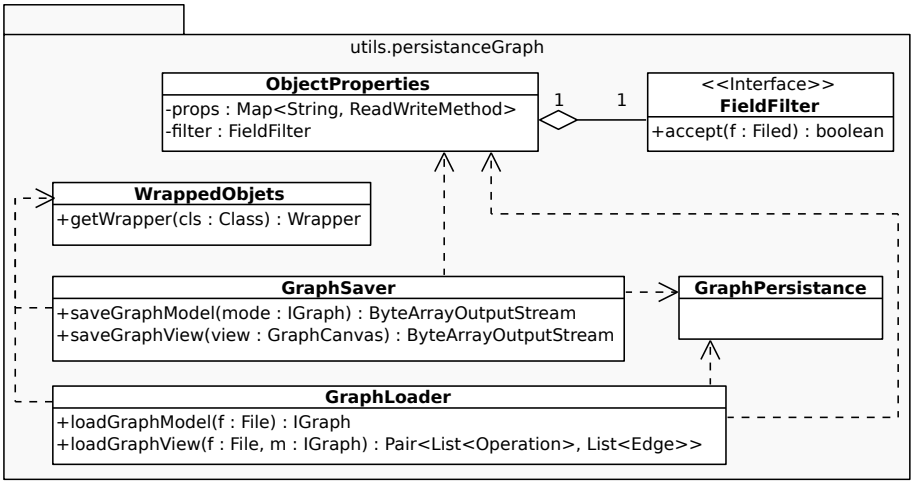
Namodelované postupy, včetně nastavení jednotlivých operací, je možné uložit a později znovu načíst. Graf je uložen ve dvou samostatných souborech. V prvním se nachází model grafu, tedy operace se svým nastavením a definované hrany. V druhém jsou uloženy informace nutné pro rekonstrukci pohledové části, to jsou pozice jednotlivých operací, pozice hran vč. souřadnic všech bodů, ze kterých je hrana složena. Před uložením jsou nakonec oba soubory zabaleny do jednoho zip archivu. Pro základní kontrolu obsahu a struktury XML souborů jsou definovány dva XSD soubory.[44]

Třídy starající se o ukládání a načítání grafu jsou v balíku `utils.persistanceGraph`, jeho struktura je vidět na obrázku 56. Základní třídy jsou `GraphSaver` a `GraphLoader`, obě poskytují dvě veřejné statické metody. Jedna slouží pro práci s modelem, druhá s pohledem. `GraphPersistance` obsahuje jména všech elementů a atributů, které mohou být použity v rámci XML souborů.

Třída `ObjectProperties` má na starost načtení všech parametrů, které mají být uloženy nebo načteny. Využívá introspekce pro prozkoumání konkrétního datového typu a na základě něj sestaví strukturu se jmény parametrů a odkazy na jejich přístupové metody. Dále obsahuje metody `get` a `set`, které získají nebo nastaví hodnotu parametru konkrétní instance daného datového typu. Jsou přeskakovány parametry označené jako

[43] Bylo ji možné vidět v diagramu na obráku 29.

[44] Pro model: `resources/graph_model.xsd`, pro pohled:`resources/graph_view.xsd`.




Obrázek 56: Struktura tříd starající se o uložení a načtení grafu.

`static`, `transient` a ty, které neprojdou filtrem. Filtr je využíván pouze při práci s operacemi, kdy jsou filtrovány pouze vnitřní vstupní parametry. U obecného objektu jsou brány všechny parametry, vyjma těch statických a neviditelných. Je tak nutné, aby pro zbývající parametry, které mají být uloženy byly definovány `get/set` metody ve formátu Java Beans. Ono obecně, aby mohla být uložena nebo načtena hodnota některého z parametrů, musí datový typ splňovat určité náležitosti. Přímo lze ukládat a načítat pouze parametry s primitivním datovým typem, textové řetězce (`String`) nebo výčtové typy `enum`. Dále je možné ukládat a načítat kolekce[45] obsahující již vyjmenované typy. Problém nastává obecně s objekty, ale i ty jsou řešeny.

K objektům se přistupuje podobně jako ke kolekcím. Základní otázka zní: *„Čím může být objekt tvořen?“*. Odpověď je, že objekt je tvořen svými parametry určujícími stav a operacemi poskytující nějakou funkcionalitu. Důležité je, že aktuální stav objektu je zachycen pomocí hodnot jeho parametrů. Jakého typu mohou být tyto parametry? Zase jen primitivní datové typy, jiné objekty nebo kolekce. Z této myšlenky vychází mechanismus a požadavky na komplexní datové typy, jejichž hodnoty mají být uloženy nebo načteny.

Nejjednodušší komplexní datové typy jsou ty, které sdružují dohromady několik primitivních typů. Aby mohl být takový objekt uložen a znovu načten musí obsahovat bezparametrický konstruktor a mít pro všechny své parametry definovány přístupové metody.[46] Dále se postupuje rekurzivně, obsahuje-li komplexní typ jeden či více parametrů, které netvoří primitivní datový typ, pak ty musí taktéž splňovat předchozí podmínky. Takovéto komplexní typy lze samozřejmě používat i v kolekcích, apod. Tímto způsobem mohou vznikat opravdu složité a komplexní datové typy, jejichž hodnoty bude možné ukládat a znovu načítat. Příklad možná až příliš složitého datového typu je ukázán v příloze E.1. Z něj je vidět, že se neomezují meze, toho co je a co není možné uložit a

[45] Typy odvozené z `Collection`, pole a slovníky (`Map`).
[46] Opět se jedná o Java Beans konvenci.

znovu načíst. Pouze je třeba u definice datových typů, které budou ukládány myslet na pár podmínek, které musí splňovat.

Co ovšem s typy, které již existují a nesplňují dané podmínky? Pro ty je definováno poslední datové rozšíření, které je reprezentováno rozhraním `Wrappper`. Pomocí něj je možné již existující typ obalit, resp. z něj vytáhnout informace, které určují jeho stav. Ty uložit a později na základě nich rekonstruovat původní hodnotu instance daného typu. Třída `WrappedObjects` pak stírá rozdíl mezi konkrétním typem a wrapperem s ním spojeným. V operacích se používají původní typy a pouze v době, kdy se hodnota dané instance ukládá nebo načítá je využit wrapper. Dokonce i v uloženém XML souboru se nachází odkaz na původní typ. Datový typ, který byl motivací pro tento přístup, byl `java.io.File`. Ten obsahuje jedinou vlastnost určující jeho stav a tím je cesta k souboru. Ovšem je možné ji nastavit pouze skrze konstruktor. Třída tak nemá bezparametrický konstruktor a metodu *set* pro její nastavení. Proto je definován `FileWrapper`, který „doplní chybějící vlastnosti“. Jeho kód je vidět ve výpisu 7.

```
public class FileWrapper implements Wrapper<File> {

    private String path;
    public FileWrapper() { }

    public void setPath(String path) { this.path = path; }
    public String getPath() { return path; }

    @Override
    public void wrap(File f) { this.path = f.getAbsolutePath(); }

    @Override
    public File unwrap() { return new File(path); }

    @Override
    public Class<File> getWrapType() { return File.class; }
}
```

Výpis 7: Příklad použití rozhraní `Wrapper`.

V kódu je vidět, že wrapper nedělá nic jiného, než že pouze vytáhne z konkrétní instance daného typu parametry určující její stav, v tomto případě cestu k souboru, definuje přístupové metody a bezparametrický konstruktor. Zbylé metody jsou implementace rozhraní `Wrapper`. Metoda `wrap` provede, již zmiňované, vytáhnutí důležitých parametrů do wrapperu. Metoda `unwrap` naopak rekonstruuje původní typ s daným nastavením. Poslední metoda `getWrapType` poskytne informaci o původním datovém typu.

Pokud je k datovému typu definován wrapper je možné na něj pohlížet jako na typ, který splňuje všechny výše zmiňované podmínky. A používat jej bez obav ve vlastních typech, které jsou používány jako vnitřní parametry operací.

### 4.7.1 Struktura XML souborů

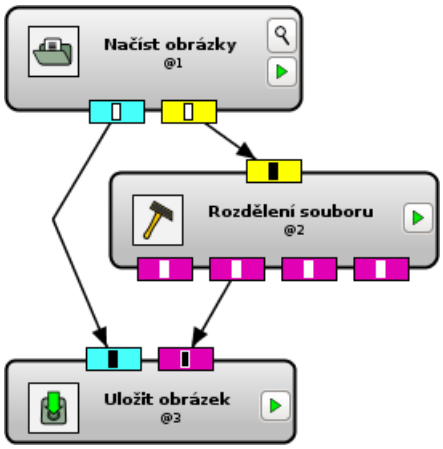
Jak pro model, tak pro pohled je základní struktura XML souboru stejná. Příklad prázdného grafu je vidět ve výpisu 8. Z výpisu je jasně patrné, že graf tvoří pouze seznam

operací a hran. Ve struktuře se ještě zhodují jména elementů pro operace a hrany, avšak v souborech s pohledem a modelem se liší v různých atributech, které dané elementy mají.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graph>
    <operations/>
    <edges/>
</graph>
```

Výpis 8: Základní struktura XML souboru popisujícího graf.

Celkově soubor s pohledovou částí je značně jednodušší než ten uchovávající model. Jelikož v pohledu se ukládají pouze informace o pozicích operací a hran. Hrany navíc ukládají pozice bodů ze kterých jsou složeny. Kdežto v modelu jsou ukládány informace o nastavených hodnotách parametrů a ty mohou být někdy dosti obsáhlé. Jak vypadají oba typy souborů je ukázáno na jednoduchém příkladu grafu, který je vidět na následujícím obrázku (obr. 57).




Obrázek 57: Jednoduchý příklad grafu.

I u takto jednoduchého příkladu jsou výsledné soubory dost rozsáhlé, proto byly přesunuty do přílohy E.2. Na závěr jen pár drobných informací k ukládání hran. Hrany v modelu jsou ukládány tak, že se uloží jméno zdrojového a cílového parametru, plus ID operace, ke které parametr náleží. V pohledové části pak může být trochu matoucí, proč je místo posledního bodu, uloženo ohraničení cílového portu. Je to z toho důvodu, že u cílových portů jsou hrany přichytávány na okraj portu, kvůli ukončení hrotem šipky. Z ohraničení portu se následně počítá souřadnice bodu, kde se hrana dotkne portu.

Díky ukládání grafů do XML souborů, je možné namodelovat algoritmus pouze ručním napsáním oněch souborů (hlavně souboru popisujícího model). To ale samozřejmě není záměr, k tomu slouží grafická nadstavba. Důvod proč jsou grafy ukládány do čitelné podoby je ten, že uživatel, který spouští graf bez grafického rozhraní někde na serveru, může chtít změnit hodnoty některým parametrům operací. Typicky to mohou být adresáře se zdrojovými daty, nebo přihlašovací údaje do databáze. Tím je umožněna jakási

primitivní editace navrženého postupu, bez nutnosti mít k dispozici, na daném stroji, nainstalováno grafické rozhraní.

## 4.8 Řešení rozšiřitelnosti

Tato kapitola shrnuje všechny typy rozšíření, které je možné do aplikace přidat. Je představen *správce rozšíření*, mající na starost jejich správu.

V předchozích částech, byly postupně přestaveny všechny druhy rozšíření, s kterými si aplikace umí poradit a zakomponovat je do sebe. Jednu třídu rozšíření tvoří dva druhy operací:

- obyčejné operace, rozhraní `IOperation` (výpis 1),
- kořenové či iterovatelné operace, rozhraní `IRootOperation` (výpis 2).

Druhou třídu rozšíření jsou tzv. typová rozšíření, které umožňují zobrazovat a editovat vnitřní parametry operací nebo přidávají novou funkcionalitu k typu (třídě), kterou původně neměl, jsou jimi:

- Rozhraní `Copier` (výpis 4), umožnující posílat data z jednoho výstupního portu do více vstupních.
- Rozhraní `ParameterCellRenderer` (výpis 5), které se stará o zobrazení (vykreslení) hodnoty parametru v tabulce vlastností.
- Abstraktní třída `ParameterCellEditor` (výpis 6), přinášející možnost editace hodnoty parametru v tabulce vlastností.
- Rozhraní `Wrapper` (výpis 7), který obalí konkrétní třídu tak, aby splňovala Java Beans konvenci a bylo možné konkretní instanci uložit do souboru a později z něj načíst.

Všechny výše zmiňované typy, je následně možné pomocí *správce rozšíření* spravovat. Ten se nachází v menu `Nástroje` → `Správce rozšíření`. Jednotlivá rozšíření jsou zveřejňovány jako samostatné `jar` balíky. Ten obecně může obsahovat libovolný počet operací a typových rozšíření.[47] Aplikace je identifikuje a nabídne k dalšímu zpracování uživateli, který je může, ale i nemusí, zpřístupnit pro práci v hlavním okně aplikace. Balík navíc může obsahovat další třídy a rozhraní, které nerozšiřují výše zmíněné typy. Mohou jimi být vlastní uživatelské typy nebo funkce, jež využívají operace či typová rozšíření. I ty jsou načteny do *classpath* aplikace.

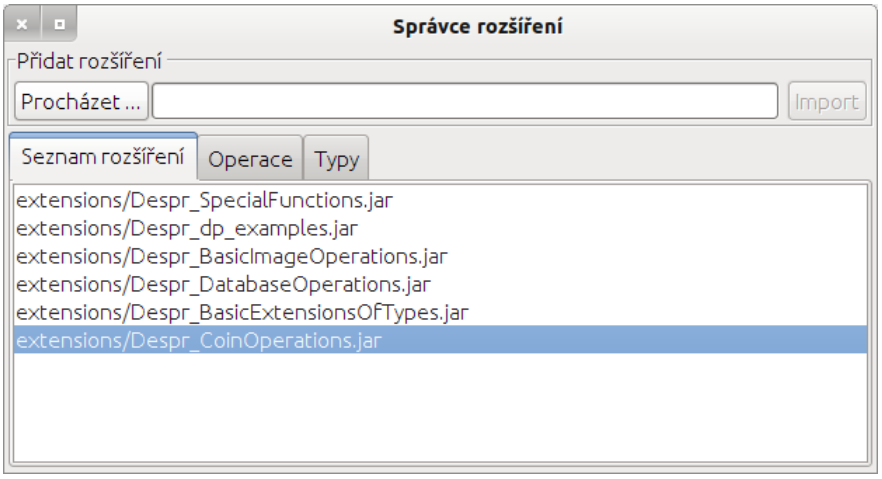
Ono přidání rozšíření není nic jiného, než že jsou z balíku vybrány všechny soubory s příponou `.class` a přidány do *classpath* tak, aby o nich aplikace věděla. Pro to je definován vlastní *class loader*, který umí za běhu aplikace *classpath* modifikovat. Všechna rozšíření tohoto typu se nachází v složce `extensions`. V adresáři s aplikací se nachází také složka `lib`. Ta slouží pro balíky (knihovny), které jsou nutné při kompilaci aplikace.

[47] Neplatí, že by jedno rozšíření muselo být v jednom `jar` balíku.

Zde se nachází např. balík s veřejným API[48] nebo ovladač k přístupu do MySQL databáze. Pokud rozšíření využívá některé z knihoven třetích stran, pak i ty je nutné zahrnout do adresáře `lib`.[49]

### 4.8.1 Správce rozšíření

Jak správce rozšíření vypadá je vidět na obrázku 58. Po spuštění se ukáže okno, kde v horní části je možno vybrat a importovat `jar` balík s novým rozšířením. Pod ním se nachází seznam všech rozšíření, které jsou v aplikaci instalována. Pomocí něj je také možné opět vybraný balík odstranit, ovšem dříve je nutné všechna rozšíření v něm obsažená „odpojit“ z aplikace.



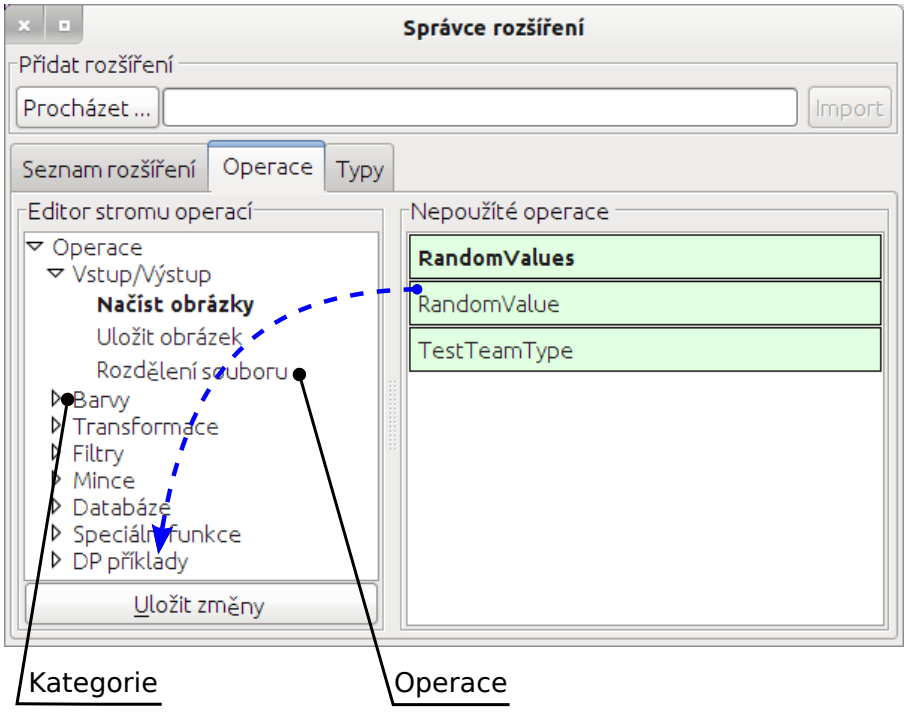

Obrázek 58: První pohled na správce rozšíření.

V záložkách jsou okna obsluhující konkrétní typy rozšíření. V prvním se nachází seznam importovaných nepoužitých operací, které lze přetažením vložit do stromu ve vedlejším panelu. V rámci tohoto okna je také možné upravovat strukturu stromu, resp. ji celou vytvořit nebo smazat. V čisté instalaci aplikace, bez jakýchkoliv rozšíření je strom prázdný. Nejsou tedy s aplikací publikovány nějaké „výchozí“ operace či typy. Záleží pouze na uživateli, které s rozšíření zvolí a nainstaluje. Vzhled záložky *Operace* ukazuje obrázek 59.

Šipka v obrázku 59 naznačuje, že jednotlivé operace se pomocí funkce *Drag & Drop* přidávají do stromu. Operace je možné vložit na konkrétní pozici ve stromu nebo přetažením na složku agregující více operací, je přidána na konec. Také jednotlivé položky ve stromu lze mezi sebou různě přetahovat, operace mezi kategoriemi či cele kategorie do jiných. Záleží pouze na uživateli, do jaké struktury si operace uloží. Jediné co je za-

[48] `despr-api.jar`

[49] Na tento fakt upozorní správce při přidávání nového rozšíření a to na základě parametru `Class-Path` v manifest souboru daného balíku.



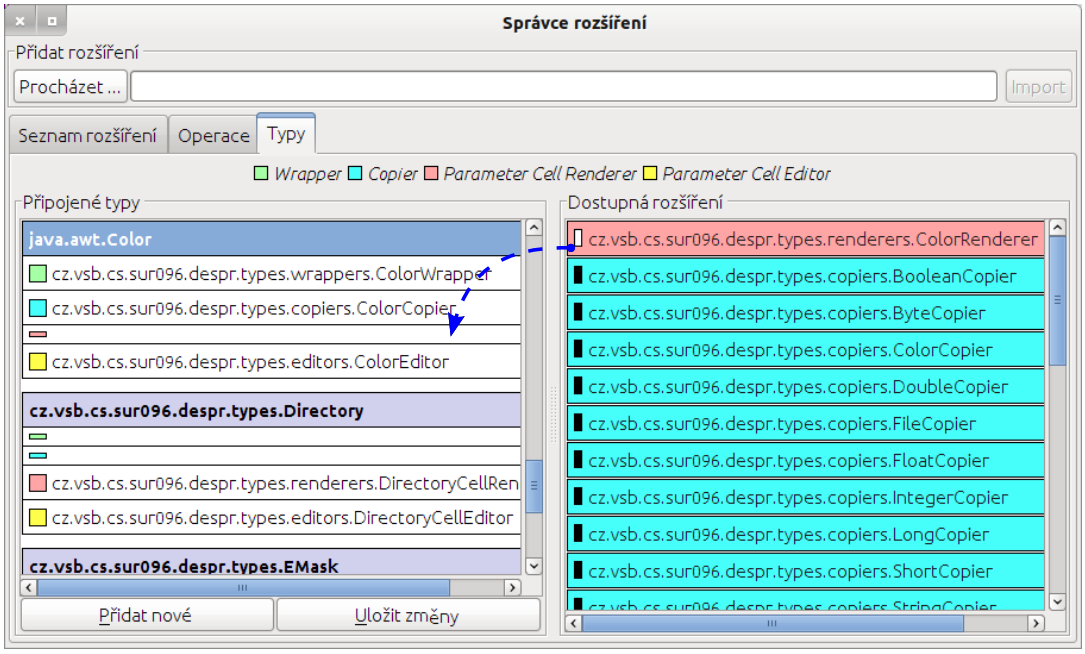

Obrázek 59: Správce rozšíření – správa operací.

kázané, je modifikace kořenové operace (`Operace`[50]), navíc do ní není možné vkládat přímo operace, pouze nové kategorie a až do nich lze přidávat operace. Je-li operace nebo celá kategorie smazána, operace v ní obsažené jsou přesunuty do panelu *nepoužité operace*. Z něj jsou smazány až po odstranění celého balíku v první záložce (*seznam rozšíření*). Jak vytvořený strom, tak seznam nepoužitých operací jsou uloženy v externích souborech v adresáři `resources`. Strom je uložen v souboru `operations.xml` a seznam nepoužitých typů v `unused_operations.list`. Tyto soubory by neměly být upravovány přímo uživatelem. Ale pro případ chyby, kdyby náhodou byl smazán balík, avšak změna se neprojevila do souborů, jsou v čitelné podobně a je tak možné odstranit operace přímo.

Poslední, třetí záložka obsahuje správu rozšíření datových typů. Jak vypadá ukazuje obrázek 60. Okno je rozděleno opět do dvou sloupců. V levém sloupci se nachází seznam jmen typů, ke kterým jsou svázána možná rozšíření. V pravém sloupci se nachází seznam všech typových rozšíření, které se nachází v některých z instalovaných balíčků.

Jednotlivá typová rozšíření jsou od sebe odlišena pomocí barev. Co která barva znamená vysvětluje legenda v horní části. Také si je možné všimnout, že vlevo od jména typového rozšíření se nachází podobná ikona jako na portech. Tato ikona má velmi podobný význam jako u portů, informuje uživatele zda je dané typové rozšíření svázané s nějakým konkrétním typem. Pokud ano, ikona je černá (typové rozšíření je použito), pokud ne typové rozšíření není svázáno z žádným typem. Značení je zde z důvodu, že

[50] Ve stromu s operacemi v hlavním okně je kořenová kategorie skryta.

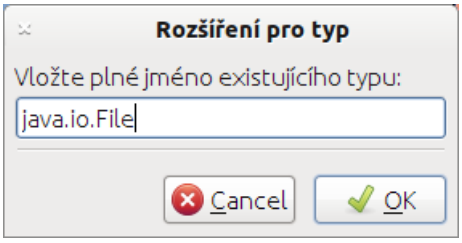



Obrázek 60: Správce rozšíření – správa typových rozšíření.

jeden typ rozšíření může být asociován na dva či více různých typů a nelze tak podobně jako u operací rozšíření ze seznamu odstranit, musí být stále k dispozici. Uživateli to také pomáhá při kontrole, kdy chce smazat celý některý z balíků v první záložce. Jak již bylo řečeno musí odstranit všechny závislost, tzn. odpojit typová rozšíření a odstranit operace ze stromu. Tak když mu správce rozšíření vypíše chybu, při mazání balíku, že je třeba ještě odpojit typové rozšíření, může zkontrolovat podle ikony zda je či není odpojené. Typové rozšíření je odpojeno, až když je odpojeno od všech asociovaných typů. Pokud tedy ikona zůstává stále černá po smazání z jednoho typu, znamená to že je rozšíření spojeno ještě z jiným typem.

Pro přidání typu, s kterým mají být spojovány jednotlivá rozšíření slouží tlačítko *přidat nový*. Zobrazí jednoduchý dialog (obr. 61) do kterého je třeba zadat jméno existující třídy, v úplném formátu tak, jak je vidět na obrázku. Je to vidět i předchozím obrázku (obr. 60), kde jsou kvůli jednoznačnosti vypisovány úplné názvy. Do dialogu lze zadávat i jména tříd nacházející se v `jar` balících. Ty nemusí ani tvořit jakékoliv rozšíření, může se jednat např. o vlastní datový typ. Při mazání balíku je pak třeba odstranit i tyto přidané typy.

Jako v předchozím případě je i seznam typů s asociovanými rozšířeními a samotný seznam rozšíření uložen v externích souborech. První se jmenuje `extensions_of_types.xml`, druhý pak `available_extensions.properties`. Oba soubory se opět nachází ve složce `resources` v adresáři s aplikací. V souboru se seznamem rozšíření je uložen i počet použití daného typu, proto *property* soubor.




Obrázek 61: Správce rozšíření – přidání nového typu.

### 4.8.2 Vytvoření vlastního rozšiřujícího balíku

Na vytvoření rozšiřujícího balíku není nic až tak zvláštního. Mělo by se jednat o standardní `jar` balík. Co je důležité je, že pokud třídy v balíku pracují s nějakými externími knihovnami[51] měly by být uloženy v adresáři `lib` na stejné úrovni jako `src`, ve kterém se obvykle nachází zdrojové soubory. Odkazy na externí knihovny by měly být uloženy v manifest souboru, s relativními cestami. Pokud by tomu tak nebylo, balík by nešel přidat.

Balík by měl být vždy distribuován s adresářem `lib`, pokud jej potřebuje[52], nejlépe zabalen v jednom archivu. Nejjednodušší způsob jak vytvořit rozšiřující balík, je založit nový projekt v Netbeans[53] načíst si veřejné API programu a normálně naprogramovat vlastní rozšíření. Jediné co pak stačí, je v panelu s projekty kliknout, pravým tlačítkem myši, na daný projekt a vybrat *build*. Do adresáře `dist` daného projektu se vygeneruje vše potřebné a nový balík je možné importovat do aplikace.

## 4.9 Jazykové mutace a vzhled aplikace

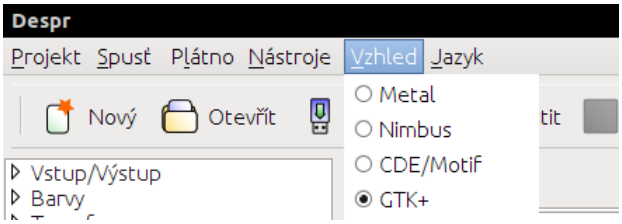
V rámci aplikace je možné provádět určitá uživatelská nastavení, konkrétně změnit vzhled a jazykovou mutaci. Změna vzhledu je víceméně kosmetickou záležitostí, avšak ne každému uživateli může vyhovovat klasický „javovské“ téma. Proto je uživateli umožněno si vybrat takový vzhled, který mu nejvíce vyhovuje. Výběr nemusí být prováděn pouze na základě vizuální stránky, ale i například ze strany rychlosti reakce prostředí, protože ne všechna vizuální prostředí reagují stejně rychle. Z tohoto pohledu nejlépe vychází téma *Nimbus*, které reaguje velmi svižně a navíc vypadá, jak v linuxu tak na windows stejně. Množství vizuálních témat, z kterých lze vybírat, záleží na konkrétní instalaci OS. Na obrázku 62 jsou např. vidět témata dostupná v Ubuntu 11.10, v němž byl nástroj vyvíjen a převážně i testován. K výběru vzhledu, ještě doporučení, je lepší vybrat na začátku jedno téma a to používat, protože výběr tématu má vliv na vzhled a hlavně rozměry operací. Grafy uložené při použití jednoho vzhledu, vypadají jinak při použití jiného[54]. Z pohledu funkčnosti to, ale význam nemá.

[51] `Jar` balíky třetích stran.

[52] Prakticky jej potřebuje vždy, protože pokud obsahuje nové operace nebo typová rozšíření, musí používat balík s veřejným API (`despr-api.jar`).

[53] Netbeans byl použit při vývoji aplikace, proto je na něj odkázáno. Bezesporu to půjde podobně jednoduše např. v Eclipse. Navíc nic nebrání autorovi rozšíření vytvořit balík jinak, např. použitím Ant skriptu.

[54] Hlavně šipky jsou posunuté a nelícují s porty, stačí však pohnout operací a hrany se korektně přichytí




Obrázek 62: Look & Feel témata dostupné v Ubuntu 11.10

Aplikace podporuje v současné době dvě jazykové mutace, **českou** a **anglickou**. Změnu je možné provést v menu, položka `Jazyk`. Aplikace jako taková bere lokalizační informace z externích textových souborů. Ty jsou uloženy v adresáři pojmenovaném ve formátu: `kód jazyka_kód země`, který se nachází ve složce `resource/lang`. O lokalizaci operací se stará autor a je na něm, jak k tomu přistoupí. Všechny reálné operace použité v aplikaci, mimo těch z ukázkových příkladů, jsou lokalizovány. Lokalizace probíhá podobně jako v případě aplikace, akorát jsou lokalizační soubory distribuovány v balíku s rozšířeními. Pokud by tedy někdo chtěl přidat další jazykovou mutaci, je to možné bez kompilace aplikace, ale je nutné přeložit všechny lokalizační soubory a těch není málo. Napsat novou lokalizaci tak může zabrat i cely den, ne-li víc. Nakonec je třeba přidat záznam o nové lokalizace do souboru `resources/Despr.properties`, ve kterém jsou uchovány uživatelská nastavení, plus seznam podporovaných jazykových mutací.

S lokalizací aplikace ještě souvisí struktura `LocalizeMessages`, která se nachází v API. Ta oproti např. `ResourceBundle` umožňuje shlukovat informace s více lokalizačních souborů. V kódu aplikace a i v kódu operací, je takto využívána v případech, kdy jedna třída rozšiřuje jinou a obě mají k sobě přidružený lokalizační soubor. Pro to, aby se nemusely v lokalizačním souboru konkretizované třídy duplikovat záznamy z nadtřídy, je možné načíst lokalizační zprávy ze dvou či více souborů do této struktury.

# 5 Zpracování obrazu

Tato kapitola tvoří úvod, k samotným metodám a navrženým postupům pro zpracování, resp. předzpracování obrazu mincí, s cílem mince vhodně uložit do databáze a s možností následného vyhledaní. Pro účely katalogizace mincí, s možnostmi vizuálního vyhledávání, tj. na základě jejich obrazu , vytvořil Ing. Petr Kašpar, v rámci své diplomové práce [3], internetový katalog. Funkční verze katalogu je dostupná na adrese `http://identifycoin.vsb.cz`.

Zpracování obrazu není proces na jeden krok. Jedná se o celou sekvenci jednotlivých kroků, které jsou aplikovány jeden za druhým, za účelem extrakce zajímavých informací z pozorované scény. Pro to byl navržen výše popsaný nástroj, který má usnadnit skládání jednotlivých operací do komplexní struktury popisující celý proces zpracování.

Zpracování obrazu začíná jeho samotným pořízením. Většinou jako první krok, po pořízení obrazu, je použití operací opravující určité nedokonalosti, jako jsou např. korekce jasu či kontrastu. Ty spadají do části, tzv. *předzpracování* (*preprocessing*).

Celý řetězec kroků vede většinou k analýze a identifikaci objektů v obraze. Jako jeden z prvních kroků v řetězci je oddělení objektů zájmu v obraze od pozadí. Pro toto se používá *segmentace* obrazu.

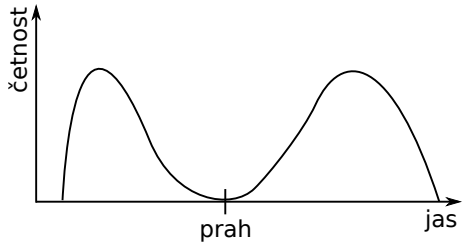
## 5.1 Segmentace

Existuje několik více či méně sofistikovaných přístupů k segmentaci obrazu. Základní rozdělení je na metody založené na detekci:

1. celých oblastí,
2. hran.

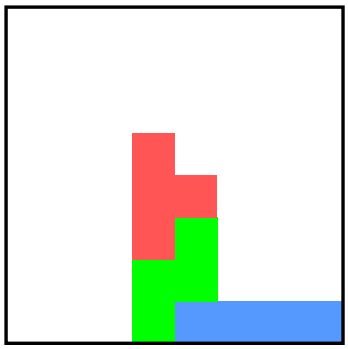
Do první skupiny metod patří i jedna z nejjednodušších metod, založená pouze na vlastnostech jednotlivých pixelů a to prahování. Princip spočívá v tom, že jednotlivé body jsou rozděleny do dvou skupin podle velikosti jasu, na ty které patří objektu (objektům) a na ty, které jsou součástí pozadí. Prahovou hodnotu lze určit na základě rozložení hodnot jasu v histogramu. Například v případě bimodálních histogramů (obr. 63) to bývá značně jednoduché a prahování dává pěkné výsledky. Např. při oddělení obrázku s textem od pozadí. Avšak v obrazech obsahující různě jasné úseky už to tak jednoduše nelze. V takovém případě je možné použít ještě adaptivní prahování, kdy hodnota prahu není určena globálně, ale pro každý pixel se počítá nová hodnota, na základě hodnot jasu v blízkém okolí.

Druhým již složitějším zástupcem metod založených na hledání celých oblastí může být metoda štěpení a spojování. Zde hraje velkou roli, tzv. kritérium homogenity. Tím může být např. jas pixelů, barva nebo i složitější funkce. V prvním kroku se obraz postupně štěpí na menší části, podle toho zda všechny pixely v dané oblasti splňují kritérium homogenity. Pokud ne oblast je zpravidla rozštěpena do dalších čtyřech podoblastí. Proces se opakuje tak dlouho dokud se štěpení nezastaví, tj. dokud všechny podoblasti nesplňují

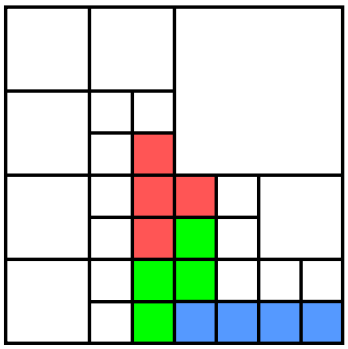



Obrázek 63: Určení hodnoty prahu z histogramu.

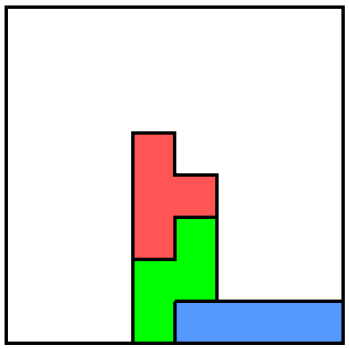
kritérium homogenity nebo pokud bylo dosaženo limitu hloubky štěpení[55]. V dalším kroku jsou postupně jednotlivé oblasti spojovány, pokud spolu souvisí a pokud opět splňují kritérium homogenity. Tímto způsobem je obraz rozdělen do vzájemně disjunktních oblastí. Příklad tohoto přístupu je postupně ukázán na obrázcích 64 až 65.



Obrázek 64: Originální obrázek



Obrázek 65: Štěpení



Obrázek 66: Štěpení – spojování

Segmentační metody založené na hranách fungují na představě, že jednotlivé objekty v obraze jsou souvislé a každý z nich je obklopen hranicí. Tento druh algoritmů je zpravidla rozdělen na dvě části. V první se naleznou informace o hranách v obraze, např. pomocí gradientních metod. Ty poskytují lokální informace, jako velikost a směr hrany v každém bodě. Druhá část procesu se pak snaží z těchto lokálních informací sestavit, nejlépe spojitou hranici, ohraničující nějaký objekt v obraze. Pro to se využívají metody jako sledování hrany nebo proložení množiny bodů přímkou či křivkou. Nebo metody založené na parametrickém popisu tvaru objektu, jako je Houghova transformace. Takto jdou v obraze detekovat např. přímky, elipsy, atp.

## 5.2 Segmentace mincí

Pro oddělení mince od pozadí je použito poloautomatického procesu při nahrávání obrázku do katalogu. Je využito právě Houghovy transformace pro detekci ohraničení

[55] Pokud je limit hloubky štěpení nastaven, nemusí se oblasti dělit až na jednotlivé pixely.

mince. Takto zjištěná ohraničení jsou nabídnuta uživateli, který z nich může vybrat to, které nejlépe obkresluje minci. Pokud ani jedno neohraničuje minci na obraze, může ji vybrat zcela manuálně, pomocí nastroje k tomu určenému. Detailněji metodu výběru mince z fotografie, je popsána v [3] - kap.3.1.

Další segmentace minci je pak triviální, protože jako vstup všech dalších algoritmů, jsou obrazy mincí vycentrované na střed obrazu a oříznuty ke svým okrajům (obr. 67). K určení ohraničení mince pak stačí vytvořit kružnici o poloměru poloviny délky strany obrazu (obr. 68). Existuje i malé procento mincí, které mají nepravidelný tvar, avšak i k těmto se přistupuje jako by byly kruhové.

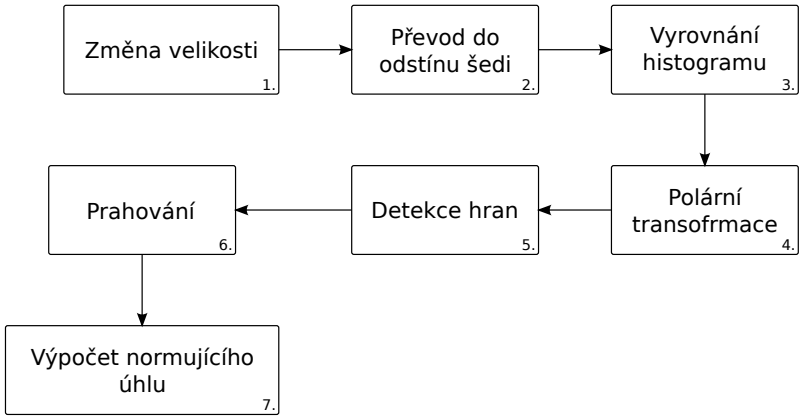

Obrázek 67: Vstupní obraz.


Obrázek 68: Výběr mince

K další segmentaci mincí již nedochází. Cílem této práce totiž není identifikovat nějaké mince na obraze a říci tady v tomto místě se nachází obraz mince. Cílem je otestovat a nalézt postupy, jak mince vhodně uložit do databáze s možností následného vyhledání. Tedy klasifikovat jejich klasifikace. Tímto směrem se také práce dále ubírá.

# 6 Aktuální stav projektu

Tato práce postupně navazuje na mou bakalářskou práci [2], ve které byly dány základy normovaní natočení obrazu mince, což byl a stále je stěžejní krok a způsob uložení obrazu do databáze. Také navazuje na diplomovou práci [3] Ing. Petra Kašpara, který vylepšil metodu normování rotace a přišel s lepším způsobem, jak minci do databáze uložit. Jednotlivé kroky stávajícího algoritmu vedoucí k výpočtu úhlu natočení ukazuje následující diagram (obr.69).




Obrázek 69: Posloupnost kroků vedoucí k výpočtu normujícího úhlu natočení.

Nejprve se provede normování velikosti obrazu na 255px. Obrazy mohou pocházet z různých zdrojů, jako fotoaparátů, skenerů, v různých velikostech. V rámci předchozích prací a testů byl stanoven tento rozměr jako ideální. Následuje převod obrazu do odstínu šedi, jelikož dále pracovat s barevnými obrázky není vůbec vhodné. Kvalita mincí může být totiž různorodá, mohou být různě zoxidované, světelné podmínky při focení mohou zkreslit barevný odstín, apod. V testovací databázi se nachází mnoho vzorů mincí, které jsou v reálu identické, avšak jejich barva na obrázku se velmi liší. Pro příklad je na obrázku 70 vidět trojice mincí, kde je vidět, že informace o barvě moc použitelná není.


Obrázek 70: Ukázka „stejné“ mince, pocházející z různých zdrojů.

Jako další krok je na obraz aplikována, tzv. polární transformace, která minci jakoby rozmotá a transformuje tak problém rotace na problém posunutí. V tomto stavu je provedena detekce hran a výsledek je ještě zprahován. Nyní se obraz nachází ve stavu, kdy jsou vyznačeny bílými (černými) pixely body, znázorňující nalezené hrany (obr. 71).

Obrázek 71: Vstup do algoritmu počítajícího normující úhel natočení, pro lepší viditelnost byla provedena inverze barev (postup byl aplikován na obrázek 67).

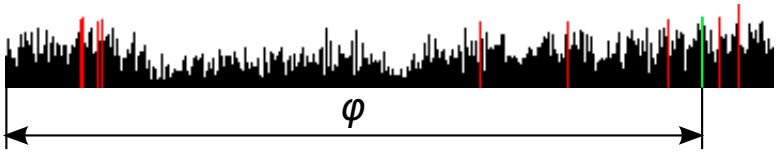
## 6.1 Normování rotace

V původní práci [2], kde se poprvé řešil problém, jak normovat natočení mince, byla nakonec použita metoda, kdy se z obrázku (podobnému obr. 71) vytvořil histogram, představující četnost bodů v jednotlivých sloupcích. Z daného histogramu se vybralo $N$ maximálních hodnot s tím, že bylo započítáno i blízké okolí, kolem daného maxima.*X-ová* souřadnice hodnoty, která po přičtení blízkého okolí byla největší, se použila jako úhel normující natočení daného obrazu mince.[56] Pro vyzdvižení, resp. potlačení, některých hodnot v histogramu, byl histogram sestavován pouze z největší spojité oblasti hranových bodů mince. Následující dva obrázky ukazují, jak byl upraven obrázek 71 (obr. 72 a jeho histogram četností (obr. 73).

Obrázek 72: Největší spojitá oblast v obraze 71, pro nalezení byla použita maska pro osmi okolí o velikosti $7 \times 7$, tzn. mezi spojitými body, může být 2px prázdná mezera.

Jak je z obrázku 73 vidět, v histogramu se často střídají malé a velké hodnoty, což znesnadňuje korektní určení normujícího úhlu. Proto bylo již tehdy počítáno s blízký okolím jednotlivých maxim, jak je psáno výše. Nicméně i s touto metodou se podařilo

[56] Jak se určuje největší spojitá oblast, resp. označení spojitých oblastí v obraze, je popsáno v [2] – kap.4.1.




Obrázek 73: Histogram četnosti hranových bodů v obrázku 72.

dosáhnout toho, že pro všechna natočení (po 1°) obrazu jedné mince, byly nalezeny průměrně 3 různá natočení, přičemž zpravidla jedno z nich dominovalo. Avšak cílem bylo dosáhnout jediného společného natočení. Protože teprve potom je možné se pokusit normovat stejné mince, resp. stejný typ, pořízených z různých zdrojů.

Zlepšení přinesl Ing. Petr Kašpar ve své diplomové práci, použitím metody tzv. *rozšířeného histogramu* [3] – kap.3.2.4. Ta se již používá pro normování mincí pořízených z různých zdrojů. Jejím základem je opět histogram četností, který se ovšem počítá z celého obrázku a ne pouze největší spojité oblasti. Tento krok se totiž ukázal být nevhodným, pro použití na obrázky pořízené z různých zdrojů a proto byl z algoritmu vynechán.

Všechny předchozí kroky jsou stejné, akorát je tedy počáteční histogram četností počítán z celého obrázku (typ obr. 71). Takto vytvořený histogram ukazuje obrázek 74.

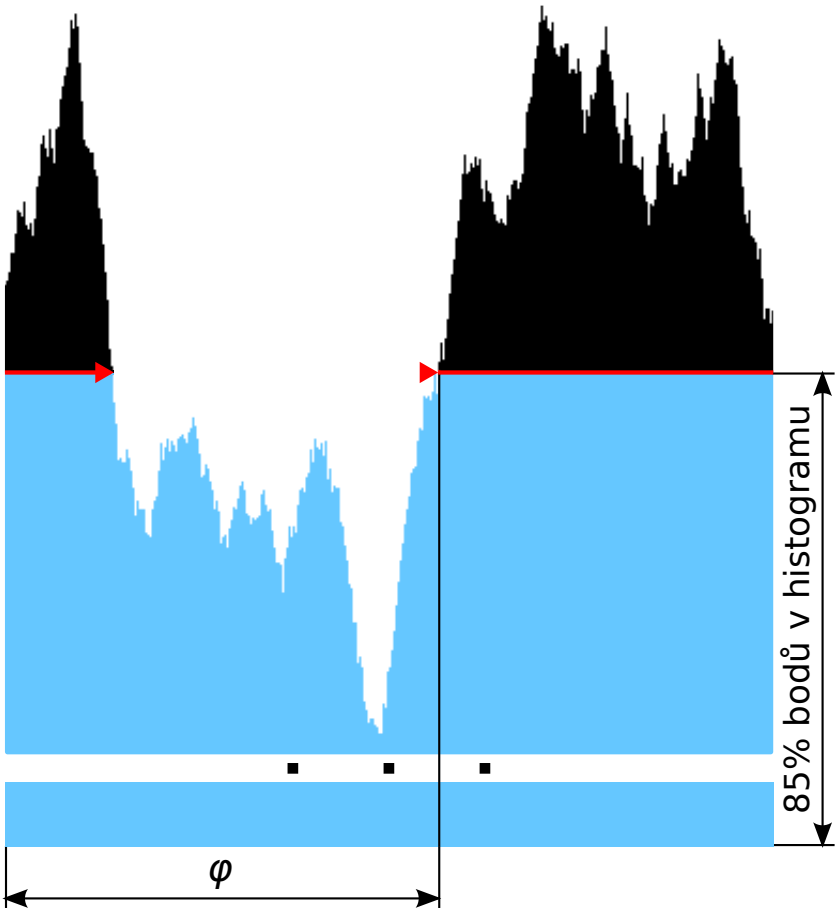
Obrázek 74: Histogram četnosti hranových bodů v obrázku 71.

Zlepšení spočívá ve „vyhlazení histogramu“, které je prováděno tak, že se ke každé z hodnot se přičtou všechny ostatní hodnoty v jejím blízkém okolí, proto *rozšířený histogram*. Následující algoritmus (alg. 3) ukazuje výpočet rozšířeného histogramu.

**Algoritmus 3** Výpočet rozšířeného histogramu.

```
1:  function GET_EXTENDED_HISTOGRAM(histogram, nearby_area)
2:      for i ← 0, histogram.length do
3:          sum ← 0
4:          for j ← i, j + nearby_area do
5:              x ← j
6:              if j ≥ histogram.length then
7:                  x ← (j − histogram.length)      ▷ Na histogram je pohlíženo jako periodickou funkci.
8:              end if
9:              sum ← (sum + histogram[x])
10:         end for
11:         extended_histogram[i] ← sum
12:     end for
13:     return extended_histogra
14: end function
```

Jak vypadá rozšířený histogram ukazuje obrázek 75. Určení úhlu nyní probíhá trochu jiným způsobem. Místo hledání maxim, je rozšířený histogram rozdělen na dvě části, kde spodní polovina tvoří 85% bodů v histogramu. Na dělící úrovni se provede řez a hledá se nejdelší spojitá oblast (červená linka v obrázku 75) a podle *x-ové* souřadnice jejího počátku se určí normující úhel $\varphi$. Během výpočtu nejdelší spojité oblasti se provádí ještě filtrování a spojování příliš malých oblastí. Detailnější popis toho, co všechno se zohledňuje je možné nalézt v [3].




Obrázek 75: Rozšířený histogram.

Metoda rozšířeného histogramu ovšem stále není stoprocentní. Dokonce sem tam vyvstávají chyby při normování různých natočení jednoho obrazu mince. Ty jsou ale vetšinou dány symetrickým vzorem na minci, který může způsobit, že mince je sice „správně“ natočena, avšak se 180° odchylkou.

# 7 Metody zpracovávající obrazy mincí a jejich testování

Tato kapitola postupně představuje metody, které byly použity pro zpracování obrazu mincí, jejich uložení do databáze, následné vyhledání a výsledky provedených testů.

Kapitola je rozdělena do čtyřech hlavních částí. V první řadě je provedeno srovnání nové verze aplikace s původní verzí, se zaměřením na srovnání výkonu a rychlosti zpracování. Dále je představen způsob testování navržených metod a popsána testovací kolekce dat. Testy jsou v tomto případě již zaměřeny hlavně na přesnost vyhledávání. V třetí části se nachází rozbor celé metody s rozšířeným histogramem vč. uložení dat do databáze a metody, resp. způsoby, jak jsou data následně vyhledávána. Jsou identifikována potenciálně slabá místa, navrženo jejich zlepšení a provedeny testy, které by měly říci zda navržené korekce vedou k lepším výsledkům či nikoliv. Poslední část popisuje odlišný přistup ke zpracování mincí. Jeho odlišnost spočívá v tom, že se nesnaží normovat natočení mince, ale pokouší se popsat obrazy mincí nezávisle na jejich natočení.

U obou hlavních testovaných metod již, nejsou podrobně rozepisovány všechny kroky algoritmu. Jsou popsána pouze místa, která jsou měněna nebo ty nově přidaná.

## 7.1 Srovnání s předchozí verzí aplikace

Srovnávat původní aplikaci (obr. 1 až 2) s novou verzí, at' již z pohledu funkcionality, uživatelského prostředí nebo práce s ní, není ani moc možné. Ve všech těchto ohledech je nová verze lepší, propracovanější a hlavně univerzálnější. Počítá se i s tím, že s ní budou pracovat další lidé a není tak omezena uzavřena na danou problematiku, i když ji je bezesporu motivována a ovlivněna. Co nová verze přináší ještě navíc, oproti té původní, je možnost paralelního zpracování operací (kap. 4.3.2). Tento fakt tak vybízí k porovnání rychlosti zpracování dat obou verzí.

Pro účely katalogu, byla naprogramována „odlehčená“ verze původní aplikace, která obsahovala pouze ty metody, jež byly použity ve výsledném postupu. Navíc byla použita účelová spouštěcí metoda, která volala jednotlivé operace tak, jak šly přesně za sebou. Tato verze měla být rychlejší verzí původního univerzálního řešení a s ní bude také porovnávána nová aplikace.

Množina obrázků, která poslouží pro tento test, bude představovat všechna natočení (po 1°) následujících čtyřech obrázků (obr. 76). Tedy rychlost bude testována celkem na 1440 obrázcích. Jedná se o stejné čtyři obrázky, které použil kolega ve své práci [3] pro otestování metody rozšířeného histogramu. Výsledky tak poslouží nejen pro porovnání obou aplikací, ale i pro ověření, že nová verze dává stejné výsledky jako ta stará.

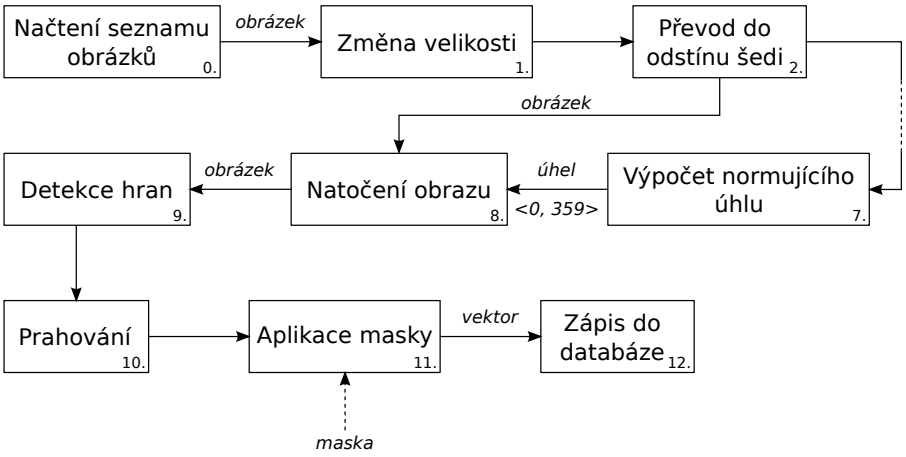
Základní posloupnost kroků, která vede k výpočtu normujícího úhlu je vidět v diagramu na obrázku 69. S tím že následující diagram (obr. 77 jej rozšiřuje a ukazuje všechny kroky původní metody. Poslední dvě metody (11. a 12.) nebyly v textu ještě nijak vysvětleny, hlavně tedy metoda *aplikace masky*.

Metoda převádějící binární obrázek na vektor čísel funguje tak, že je přes obraz mince přeložena maska a v každém ze segmentů je spočítán počet hranových bodů (bílých pixelů). Nejedná se o obyčejné čtvercové masky jaké byly použity v [2], ale o masky zohledňující tvar mince. S nimi poprvé přišel kolega v rámci své práce. Podrobnější popis

Obrázek 76: Základní vzory, ze kterých je vygenerována testovací kolekce 1440 obrázků.

je možné nalézt [3]-kap.4.2.2. Maska která je použita při testování je vidět na obrázku 78, obsahuje 69 segmentů.

Metoda ukládající vektor do databáze, navíc ukládá i jméno obrázku, cestu ke zdroji a úhel, který byl použit pro natočení. V původní verzi byla, každá hodnota vektoru uložena v jednom sloupci tabulky. V aktuální verzi, je vektor ukládán jako textový řetězec, kvůli kterému byly doprogramovány dvě nativní metody, jedna pro výpočet vzdálenosti mezi vektory a druhá pro jejich porovnání. Blíže o nich v příloze D zabývajícími se popisem jednotlivých operací. Nachází se zde i návod, jak metody do databáze nainstalovat.




Obrázek 77: Původní metoda předzpracování obrazů mincí.

Nyní zbývá již pouze říci, jaké je nastavení jednotlivých operací v původní metodě. U operací jsou uvedeny pouze klíčové parametry, které ovlivňují výsledky zpracování. Operace které žádné parametry neobsahují a operace ukládající výsledky do databáze nejsou ani uvedeny ve výpisu. Parametry použité v původním řešení:

**Změna velikosti,** všechny obrázky jsou normovány na velikost $255 \times 255$px.

**Polární transformace,** šířka obrázku po polární transformaci je nastavena na 412px.

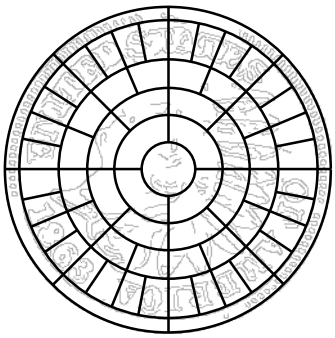
**Detektor hran,** pro detekci hran je použit *Cannyho hranový detektor*, s následujícím nastavením:

- Velikost masky: 3,
- úhel: 0.45 (25.8°)
- spodní práh: 6,
- horní práh: 25,
- scale: 1.0,
- offset: 0.

**Prahování,** hodnota prahu je nastavena na 25.

**Výpočet normujícího úhlu,** tato operace obsahuje dva parametry. Prvním je procentuální určení, které rozdělí rozšířený histogram na dvě poloviny (obr. 75), ten je nastaven na hodnotu 85%. Druhý parametr určuje minimální velikost nejdelší spojité oblasti, ten je nastaven na hodnotu 10.

**Aplikace masky,** jak již bylo řečeno tato metoda používá masku, pomocí které převede binární obraz na vektor. Jedním z parametrů je samotný typ masky, ta je vidět na obrázku 78, druhým parametrem je pak informace o tom, zda mají být jednotlivé hodnoty vektoru ukládány v absolutních hodnotách nebo v relativních (procentuální zastoupení hranových pixelů v dané oblasti). Ve vektoru ukládaném do databáze se nachází absolutní hodnoty.



Obrázek 78: Maska s 69 segmenty.

### 7.1.1 Testovací hardware

Všechny testy byly prováděny na notebooku HP6510b.

- Procesor Intel Core 2 Duo T7500,
- operační paměť 3GB DDR2 667 MHz,
- Pevný disk 250GB, 5400 RPM,
- OS Ubuntu 11.10/32bit, DB: MySQL v5.1.62.

### 7.1.2 Výsledky testů

Byly provedeny dva testy. V prvním se ukládaly téměř všechny mezivýsledky, konkrétně u operací: 1, 2, 3, 4, 5, 6, 8 (obr. 69 a 77). V druhém případě se ukládal, pouze výsledek operace 8. (natočení obrazu). Důvod k tomuto oddělení je srovnání paralelní a sériové verze. Ukládání mezivýsledků je totiž v nové verzi prováděno paralelně s dalšími operacemi.

Hlavním cílem těchto testů je srovnání rychlostí obou aplikací a porovnání výsledků, tzn. že obě verze by měly, při stejném nastavení parametrů, poskytnout stejné výsledky. Všechny testy byly prováděny již na zmiňované kolekci 1440 obrázcích. Oba dva namodelované testy je možné nalézt na přiloženém CD v adresáři: `prilohy`, `korekcni_test_1` a `korekcni_test_2`.

#### 7.1.2.1 První srovnání

Výsledky prvního srovnání jsou více než překvapivé:

- Stará verze, doba zpracování: 1 hodina, 1 minuta a 23 sekund.
- Despr, doba zpracování: 13 minut a 43 sekund.

Takovýto rozdíl v časech nebyl v žádném případě vůbec očekávaný a není ani důvod k takovému velkému zrychlení. Výsledek naznačuje jediné, buď se v první verzi počítá něco navíc nebo je zde obsažena chyba. Po bližším prozkoumání se ukázalo že jsou pravdivé obě varianty. Hned v první chvíli bylo zjištěno, že se nepočítá jedna verze algoritmu, ale hned tři. Místo detekce hran a prahování, jsme tehdy ještě zkoušeli navíc variantu s adaptivním prahováním obrazu mince a metodu, kdy se obraz ponechal v odstínech šedi a vypočetla se průměrná hodnota jasu v daném segmentu masky. Obě tyto metody nedosahovaly valných výsledků. Po vypnutí těchto výpočtů byl čas zpracování staré verze aplikace: **29 minut a 23 sekund**, kterému by už se dalo věřit, avšak druhé srovnání ukazuje, že tomu tak není.

#### 7.1.2.2 Druhé srovnání

Při provedení druhého srovnání, kdy se ukládaly pouze výsledky po normování natočení, ovšem vyšly opět nečekané výsledky:

- Stará verze, doba zpracování: 21 minut a 22 sekund.
- Despr, doba zpracování: 10 minut a 19 sekund.

V obou případech se totiž jedná o téměř sériové zapojení operací a dvě operace, které jsou v nové verzi prováděny paralelně by neměly dvojnásobně zrychlit výpočet. Tentokrát se opět jednalo o výpočet navíc, který může mít ovšem neblahé následky na výsledky vyhledávání a jedná se tak fakticky o chybu v aplikaci.

Původní aplikace se totiž skládá ze dvou částí (verzí). První slouží pro předzpracování celé kolekce dat a vytvoření databáze. Druhá verze se pak používá pro transformaci jednoho obrazu mince na vektor, podle kterého se následně vyhledává mince v databázi. Problém je v tom, že obě dvě verze provádí normování natočení s tím, že první verze

využívá metodu té druhé, která provádí transformaci obrazu mince na vektor. Ve skutečnosti se tak provádí normování natočení $2\times$, jedno na vstupní minci, podruhé již na tu normovanou. Díky tomu, že metoda normování natočení není stoprocentní, může tento fakt dělat ve vyhledávání problémy.

#### 7.1.2.3 Srovnání po odstranění chyby

výsledky po odstranění chyb jsou následující:

**První test:**

- Stará verze, doba zpracování: 22 minut a 5 sekund.
- Despr, doba zpracování: 13 minut a 43 sekund.

**Druhý test:**

- Stará verze, doba zpracování: 14 minut a 25 sekund.
- Despr, doba zpracování: 10 minut a 19 sekund.

Dosažené výsledky již více odpovídají realitě. Sice jsou stále vidět nezanedbatelné rozdíly v obou testech, avšak to může být dáno různou efektivitou kódu, jak prostředí které spouští jednotlivé operace, tak samotných operací. Během přípravy nové verze aplikace, byly totiž všechny původní operace přepsány a při tom byla snaha jejich kód optimalizovat, pokud to bylo možné.

Co je zajímavé, je porovnání obou běhů staré verze mezi sebou a stejně tak u nové. V prvním případě je rozdíl 7 minut a 40 sekund, v druhém pak 3 minuty a 24 sekund. Zde je vidět jasné zlepšení paralelního přístupu a při šikovném namodelování úloh v nové verzi, lze výrazně zkrátit dobu zpracování. Pro zajímavost byl ještě otestován běh nové verze v prostředí bez grafické nadstavby (prostředí serveru). V tomto případě byly sice časy o něco málo lepší, ale už pouze zanedbatelně.

Nakonec byly porovnány výsledky obou aplikací. Ty dopadly dobře a shodovaly se. Při přepisování operací, tak nebyly do jejich kódu zaneseny další chyby. Do tohoto srovnání se naštěstí výše zmiňovaná chyba nepromítla, protože uložení obrázku proběhlo po první aplikaci normování rotace. V tuto chvíli je možné provádět další testovaní s tím, že bude možné srovnávat nové výsledky z těmi, kterých dosáhl Ing. Petr Kašpar a říci zda se podařilo metodu vylepšit či nikoli. Také bude proveden referenční test se stejným nastavením, jako v těchto testech, který by měl ukázat zda chyba nalezena ve staré verzi aplikace měla vliv na konečné výsledky. Navíc referenční test podá velmi dobrou informaci o metodě normující natočení mince, jelikož i s touto chybou dosáhl kolega docela zajímavých výsledků.

## 7.2 Metodika testování a popis testovací kolekce dat

Testování bude prováděno velmi podobným způsobem jako v práci [3] – kap.8. Z celé kolekce téměř dvou set tisíc obrázků mincí je pro testy náhodně vybrána kolekce dvaceti pěti tisíc obrázků. Z ní je pak vybráno osmdesát vzorů, které poslouží jako vyhledávací

množina. Navíc oproti předchozí práci, je vybráno dvacet obrázků mincí, které se v testované množině přímo nenachází. Jedná se o obrázky, na kterých se nachází stejný typ mince. Ve skutečnosti se tak jedna o jinou minci, např. s jiným letopočtem, focenou nebo skenovanou jiným přístrojem, za jiných světelných podmínek, atd.

V první fázi vždy proběhne předzpracování celé kolekce danou metodou a vytvoření databáze. Následně je provedeno vyhledání s použitím dané metody, s tím že mince ve vyhledávací množině jsou otočeny o náhodný úhel, kvůli korektnímu otestování celého postupu, vč. normování natočení. Vyhodnocení pak probíhá tak, že je při hledání vybráno prvních deset nejlepších výsledků a zjišťuje se, zda se mince v TOP 10 objevila a pokud ano, tak na kterém místě. Celkově je test považován za úspěšný, pokud se mince ve výsledcích objeví do desáté pozice včetně.

### 7.2.1 Testovací množina

Všechny obrázky v testovací kolekci jsou připraveny pro předzpracování a uložení do databáze. Obrázky mají různé rozlišení, nejčastěji se nachází mezi, cca. $280 \times 280$ px a $600 \times 600$ px, jsou vycentrovány na střed a oříznuty. Není tak třeba minci na obrázku hledat. V reálu se o tento problém stará poloautomatický systém na vstupu katalogu. Metody použité pro předzpracování a vyhledávaní počítají s tím, že vstupní data jsou v korektní podobě.

Pro příklad následující dvě čtveřice obrázků (obr. 79) ukazují, v jak rozdílných kvalitách se mohou mince nacházet. Toto není problém fotografie avšak samotného stavu mince. Jednotlivé mince mohou být různě zkorodované, potlučené, atd.


Obrázek 79: Ukázka toho, jak mohou vypadat data v testovací množině.

### 7.2.2 Vyhledávání

Pro vyhledávaní se používá prakticky stejný SQL dotaz jako v [3] – kap. 5.2.4, akorát využívá dvou dodefinovaných funkcí v MySQL databázi a to: `compute_distance` a `compare_vectors`. První spočítá vzdálenost dvou vektoru[57]. Druhá porovná vektory s tím, že řekne pouze, zda jsou či nejsou vektory podobné. Aby si byly vektory podobné, nesmí se jednotlivé položky na stejných pozicích lišit o $\pm c, c \in \mathbb{N} \wedge c \geq 0$.

Pro účely vyhledávání mincí v databází se v aplikaci nachází metoda *hledej minci*. Ta v sobě skrývá onen SQL (výpis 9) dotaz a jako jeden z parametrů, je *relativní prah*, který představuje zmiňovanou hodnotu $c$. Druhý parametr, který ovlivňuje přesnost hledání je *absolutní prah*, který říká, že součet všech hodnot hledaného vektoru se nesmí lišit o $\pm$*absolutní prah* se součtem všech hodnot porovnávaného vektoru. Neboli musí platit, že:

$$\left| \sum_{i=0}^{N} v_i^{DB} - \sum_{i=0}^{N} v_i^{S} \right| \leq abs\_threshold \tag{1}$$

kde index *DB* znamená vektor uložený v databázi a *S* hledaný vektor. Při vyhledávání jsou nejprve vektory odfiltrovány podle vztahu 1 a po té porovnány metodou `compare_vectors`. Nakonec je množina seřazena podle vzdálenosti a vybráno několik prvních reprezentantů. Pseudokód dotazu ukazuje následující výpis (výpis 9).

```
SELECT *, abs(sum(v_db) − sum(v_s)) as total_tolerance, compute_distance(v_db, v_s) as distance
  FROM tableName WHERE
    total_tolerance  < abs_threshold AND compare_vectors(v_db, v_s)
  ORDER BY distance, total_tolerance
  LIMIT 0, max;
```

Výpis 9: Pseudokód SQL dotazu pro vyhledávání.

## 7.3 Metody založené na rozšířeném histogramu a jejich výsledky

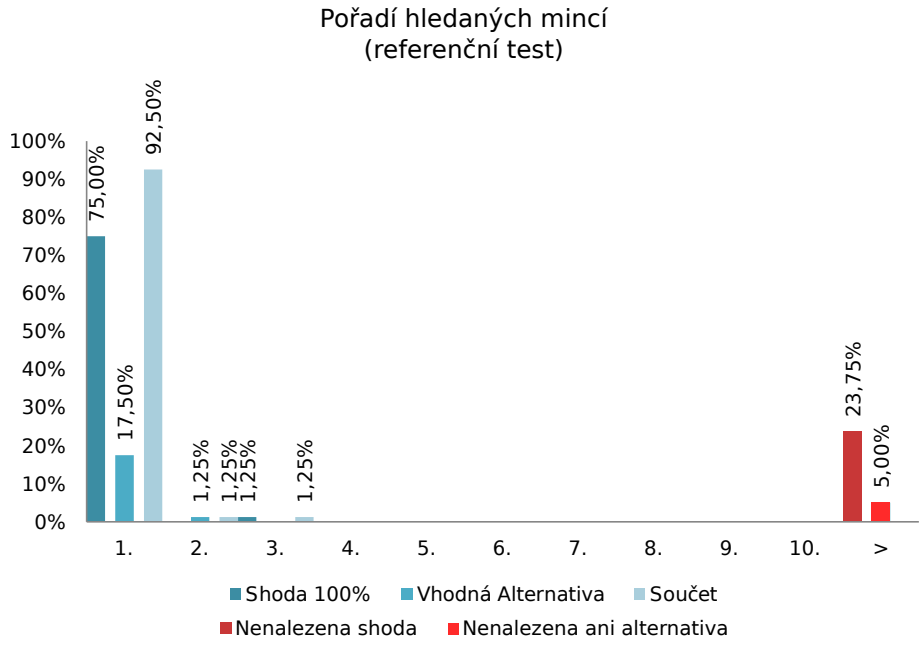
V této kapitole jsou sepsány výsledky testů metod, založených na metodě rozšířeného histogramu. Jako první je proveden referenční test, se stejným nastavením jako ve srovnávacích testech s tím, že byla odstraně nalezená chyba. Další testy založené na této metodě se již liší v nastavení parametrů některých operací či modifikací postupu, za použití jiných metod. Důvody změn jsou vždy rozebraný na začátku každého z testů.

### 7.3.1 Výsledky referenčního testu

Výsledky referenčního testu po odstranění nalezené chyby, dopadly velmi dobře. Kolegou získané výsledky dosahovaly $65\%$ úspěšnosti nalezení hledané mince na první pozicí. Po započtení nalezených výsledků do desáté pozice se úspěšnost zvedla $78.5\%$. Navíc byla, jako alternativa, při neúspěchu použita metoda využívající tzv. masku s přesahy (více v [3] – kap.4.3.7). Té se minci podařilo nalézt v $2.5\%$ případů, ale již není uvedeno na jaké pozicí. Celkovou úspěšnost Ing. Petr Kašpar udává $\mathbf{81}\%$.

[57] Je možné vybrat mezi euklidovskou nebo city block distance, pro testy se používá euklidovská vzdálenost

Nové výsledky referenčního testu výrazně předstihly ty předchozí a ne jen co se týče úspěšnosti, ale i přesností. Výsledky jsou shrnuty v tabulce 1 a vyneseny do grafu (obr. 80), poslední dvě hodnoty odlišeny červenými barvami se nesčítají. První vyznačuje chybovost při hledání 100% shody. Druhá udává počet mincí, u kterým nebyla nalezena ani alternativa na prvních deseti pozicích.




Obrázek 80: Výsledky vyhledávaní referenčního testu.

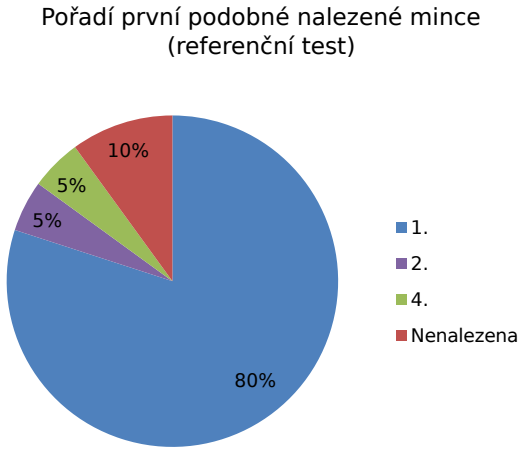
| | **Pořadí** | 1. | 2. | 3. | 4. | 5. | 6. | 7. | 8. | 9. | 10. | > |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Absolutní hodnoty** | **Shoda** | 60 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 19 |
| | **Alternativa** | 14 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | **Součet** | 74 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | – |
| | **Celkem** | 76 | | | | | | | | | | – |
| **[%]** | **Shoda** | 75 | 0 | 1.25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23.75 |
| | **Alternativa** | 17.5 | 1.25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| | **Součet** | 92.5 | 1.25 | 1.25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | – |
| | **Celkem** | 95 | | | | | | | | | | – |

Tabulka 1: Výsledky vyhledávání referenčního testu.

Jak je z výsledků vidět, oprava chyby v původním programu vedla k dosti velkému zlepšení. Mince jsou většinou nalezeny hned na první pozici nebo nejsou nalezeny vůbec. V takovém případě se ovšem ve 14ti případech objevila na prvním místě alternativní mince, tzn. stejná mince akorát z jiného zdroje. To znamená 92.5% úspěšnost nalezení mince hned na prvním místě. Mince se také našly v ojedinělých případech na dalších pozicích. Celkově tak s celé kolekce, osmdesáti hledaných vzorů, nebyly nalezeny pouze

4 mince, což je 5% nespěšnost. Neúspěšnost 23.75% znamená neúspěšné hledání, co se týče stoprocentní shody. Nakonec když se sečnou výsledky na všech pozicích bylo dosaženo celkové 95% úspěšnosti.

U druhého testu s obrázky, které nebyly přímo uloženy v databázi dopadly výsledky také velmi dobře. Jak ukazuje druhý graf (obr. 81) podobná mince byla nalezena v 80% hned na prvním místě. Dvě mince byly nalezeny na jiných pozicích a dvě nebyly nalezeny vůbec. Je také zajímavé, že když už byla podobná mince nalezena, tak většinou nebyla jediná. V 50% případů se dokonce podobná mince objevila na všech deseti pozicích. Ostatně jaké byly počty nalezených podobných mincí, ukazuje třetí graf (obr. 82). Z těchto výsledku plyne, že navržená metoda je vcelku robustní a zvládá korektně zařazovat i obrázky, které nejsou přímo uloženy v databázi.



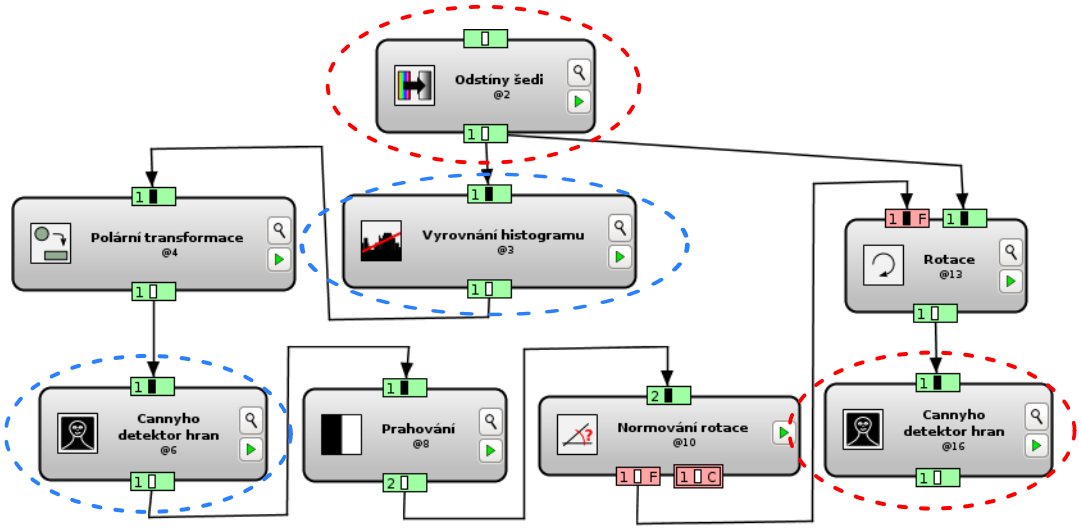

Obrázek 81: Procentuální rozložení pořadí, nalezení první podobné mince (test 1).


Obrázek 82: Celkové počty nalezených podobných mincí (test 1).

## 7.4 První část modifikací referenční metody

První modifikace spočívá v jedné úpravě namodelovaného postupu a v jedné změně parametru. U změny parametru, se jedná o změnu šířky obrázku po polární transformaci. Původní hodnota 412px byla stanovena tak, aby obsah obrázku po polární transformaci byl přibližně stejný jako obsah kruhu o poloměru 127px. Nicméně z výsledků různých malých testů, zaměřených na normování natočení se zdá být lepší hodnota 360px. Výsledné normované obrázky jsou pak více stabilnější a „nekmitají“ tolik. U původní metody sice obrázky na první pohled vypadaly, že jsou normovány na stejný úhel, ale po podrobnějším prozkoumání se jednotlivé výsledky drobně liší a v některých případech i více, jak o $5°$. Tento fakt je možné zdůvodnit tím, že jako normující úhel se používá *x-ová* souřadnice nalezeného maxima v rozšířeném histogramu (kap. 6.1), který vzniká z obrázku po polární transformaci. Při použití šířky 360px je histogram o 52 pixelů (sloupců) užší a je tak menší pravděpodobnost vzniku drobné odchylky. Navíc se při tomto rozměru nemusí přepočítávat úhel do rozsahu $0°$ až $359°$. Z těchto důvodů byl zvolen a testován, v dalších metodách, rozměr 360px.

Druhá změna vyplývá z jisté nekonzistence posloupnosti některých kroků. Konkrétně, se $2\times$ aplikuje hranový detektor se stejným nastavením, jednou na obraz po polární transformaci a jednou na původní obraz po normování natočení, jenže vstupní obrazy jsou v těchto krocích různé. Tento fakt byl odhalen použitím nové verze aplikace, kde je možné si téměř okamžitě všimnout, že něco není v pořádku.




Obrázek 83: Vyznačení nekonzistence v původní metodě.

Na obrázku 83 je vidět posloupnost bloků s vyznačením toho, co nesedí. Jedna detekce hran je aplikována na obraz, u kterého bylo již před tím provedeno vyrovnání histogramu, ovšem ta druhá se aplikuje na obraz, který byl pouze převeden do odstínu šedi. Detektor hran by měl být, v druhém případě, použit až na obraz po vyrovnání histogramu, nikoli předtím. Proč tomu tak není ukazuje obrázek 84. Jak je z něj vidět, použitím klasické

metody vyrovnání histogramu na celý obraz, vzniknou okolo mince nežádoucí artefakty, které negativně ovlivní detekci hran. Tehdy, asi kvůli jednoduchosti, byl použit obrázek pouze po převodu do odstínu šedi a postupem času se na to pozapomnělo. Nyní je tento rest napraven a místo klasické metody vyrovnání histogramu je použita metoda, která danou operaci provede pouze na oblasti mince. Výsledek je možné vidět na obrázku 85. Na první pohled si je možné všimnout znatelného zlepšení, ne jen co se týče vzniku artefaktů, ale celkově je obrázek více prokreslen. To naznačuje, že i hrany by mohly být lépe detekovány a tím pádem by mohla celá metoda poskytovat lepší výsledky.


Obrázek 84: Výsledek klasického vyrovnání histogramu.

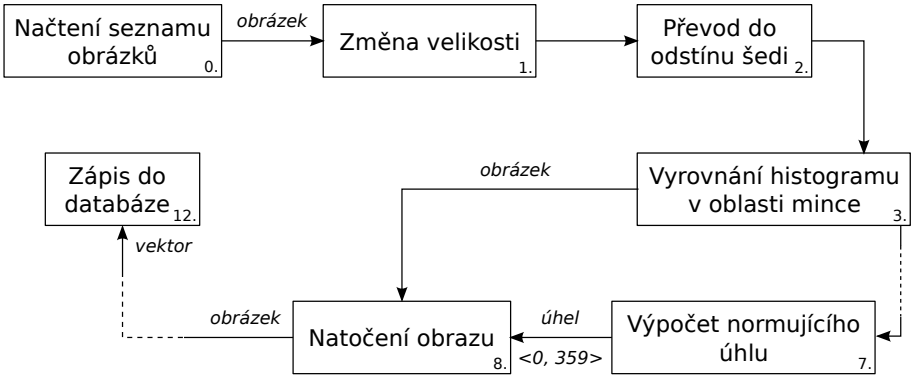

Obrázek 85: Vyrovnání histogramu aplikovaného pouze na oblast mince.

### 7.4.1 Nastavení metod

S daným nastavením byly provedeny dvě verze testů. První pouze se změnou popsanou výše. Diagram na obrázku 86 ukazuje, jak se metoda liší od původní verze (obr. 77). Je v něm vidět, že byl vyměněn blok vyrovnávající histogram obrazu, na jehož výsledky je později aplikována detekce hran (v obou případech tak na stejný typ vstupů). Také byla změněna šířka obrazu po polární transformaci na 360px, kvůli větší stabilitě normovaných obrázků. Zbytek zůstává stejný.

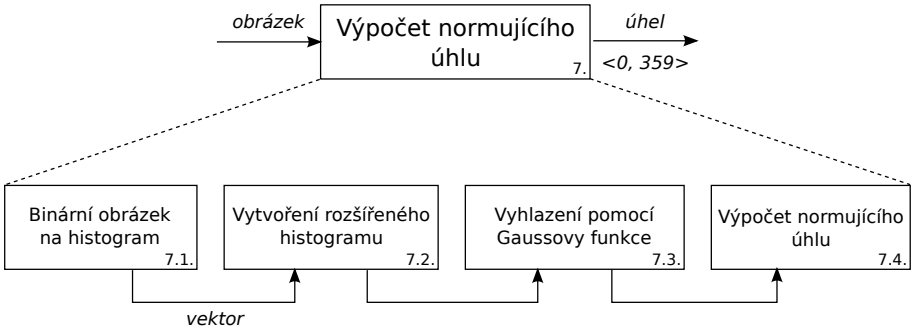
V druhé verzi je navíc vyhlazen rozšířený histogram za pomoci Gaussovy funkce (vzorec. 2). Pro tyto účely byl původní blok 7. (*Výpočet normujícího úhlu*) nahrazen sekvencí kroků (obr. 87), kdy první blok (7.1) vytvoří z binárního obrázku histogram, druhý blok (7.2) vytvoří rozšířený histogram (alg. 3). Ten je nyní vyhlazen pomocí Gaussovy funkce s nastavením $\sigma = 2$ (blok 7.3) a až z tohoto výsledku je počítán úhel natočení (blok 7.4). Samotné vyhlazení se provádí konvolucí vektoru s histogramem a maskou, která je vygenerována pomocí vzorce 2.

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$




Obrázek 86: Úprava metody po výměně bloku vyrovnávající histogram obrazu.

Poslední metoda (blok. 7.4) má stejné parametry jako ta původní, tedy procentuální vyjádření plochy, která rozdělí histogram na dvě poloviny a prahovou hodnotu odstraňující příliš malé spojité oblasti. První z hodnot byla zvýšena z 85% na 90%, jelikož vyhlazení histogramu způsobí určitý pokles a histogram je tak nutné „říznout“ ve vyšší části, aby nevznikla jedna velká spojitá oblast, přes celý obrázek, čímž by metoda úplně selhala. Druhá hodnota zůstává stejná, tedy 10.

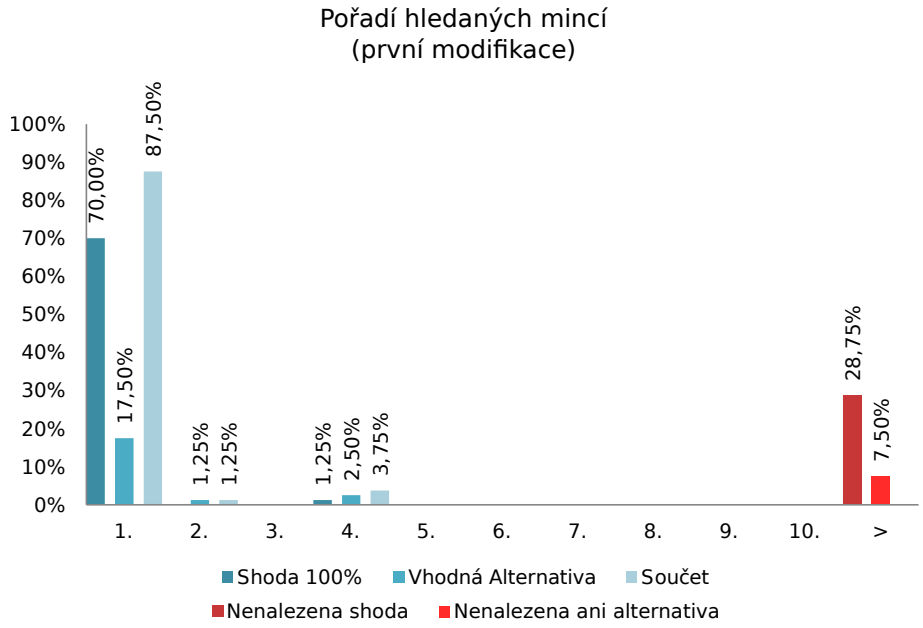



Obrázek 87: Vyhlazení rozšířeného histogramu.

### 7.4.2 Výsledky testu první modifikované metody

Výsledky vyhledávání první modifikované metody (graf na obr. 88) se na první pohled mohou zdát horší než u výsledku referenčního testu. Co se týče celkové úspěšnosti tak ta klesla jen o 2.5%. Horší je to v případě schopnosti hledat mince na první pozici kde je 5% pokles. Zvedl se však podíl mincí nalezených na dalších pozicích, což ztrátu jistým způsobem kompenzuje.

Pozitivní efekt této změny se projevil v druhé verzi testu, kdy jsou hledány podobné mince. Nyní nebyla podobná mince nalezena, pouze v jediném případě. A celkový počet nalezených podobných mincí je 149 z 200 možných. Oproti původní metodě, se nejen zvýšil počet podobných kusů (sice nepatrně), ale i přesnost. Kromě jedné nenalezené

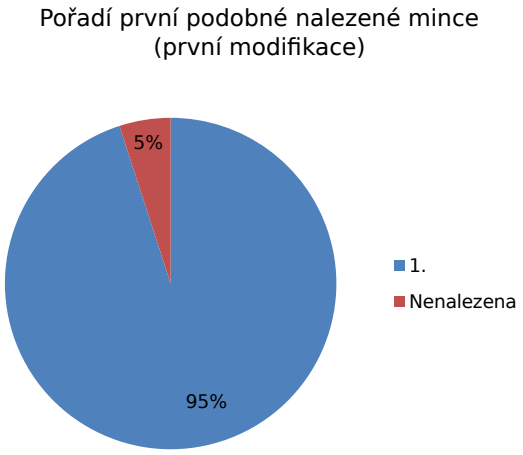



Obrázek 88: Výsledky vyhledávaní první modifikované metody.

mince, byly v ostatních případech nalezeny podobné mince vždy na prvním místě (graf na obr. 89).

| | **Pořadí** | 1. | 2. | 3. | 4. | 5. | 6. | 7. | 8. | 9. | 10. | > |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Absolutní hodnoty** | **Shoda** | 56 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 23 |
| | **Alternativa** | 14 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |
| | **Součet** | 70 | 1 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | – |
| | **Celkem** | 74 | | | | | | | | | | – |
| **[%]** | **Shoda** | 70 | 0 | 0 | 1.25 | 0 | 0 | 0 | 0 | 0 | 0 | 28.75 |
| | **Alternativa** | 17.5 | 1.25 | 0 | 2.5 | 0 | 0 | 0 | 0 | 0 | 0 | 7.5 |
| | **Součet** | 87.5 | 1.25 | 0 | 3.75 | 0 | 0 | 0 | 0 | 0 | 0 | – |
| | **Celkem** | 92.5 | | | | | | | | | | – |

Tabulka 2: Výsledky vyhledávání první modifikované metody.



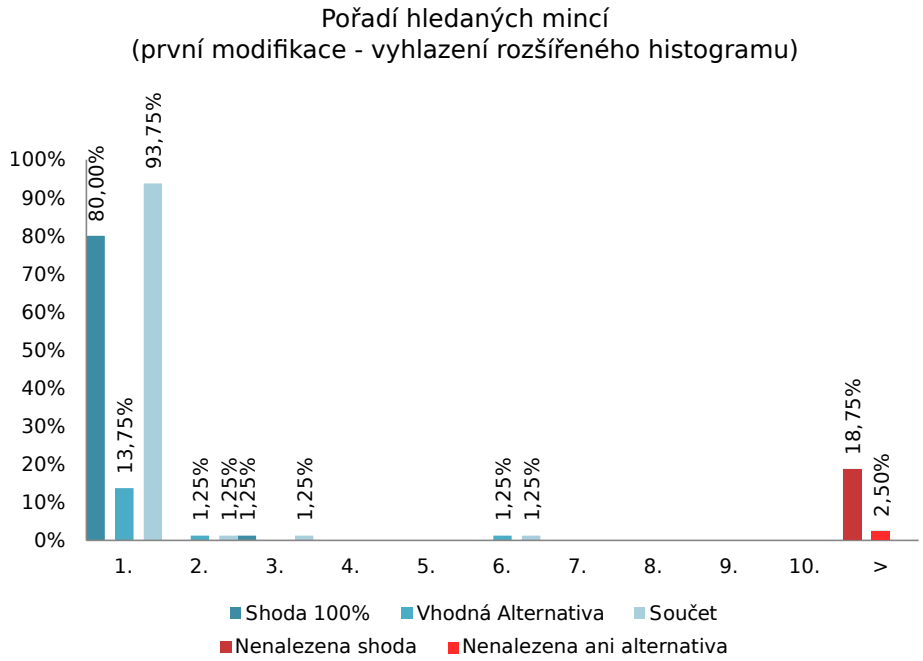

Obrázek 89: Procentuální rozložení pořadí, nalezení první podobné mince (test 2).


Obrázek 90: Celkové počty nalezených podobných mincí (test 2).

### 7.4.3 Výsledky vyhledávaní metody s vyhlazeným rozšířeným histogramem

Přidáním metody, která před výpočtem normujícího úhlu, ještě vyhladí rozšířený histogram bylo dosaženo vůbec nejlepších výsledků, alespoň co se týče první části testu. Jak je vidět v tabulce 3, hned na prvním místě bylo nalezeno 64 mincí, které se stoprocentně shodovaly s hledanými vzory. Jen toto číslo znamená 80% úspěšnost hledání, což je v porovnání s původním s výsledky z [3] velmi pěkný výsledek, kdy bylo dosaženo maximální celkové úspěšnosti 81%. Po přičtení nalezených alternativ a mincí umístěných na dalších pozicích je celková úspěšnost této metody 97.5%.

Na druhou stranu zlepšení výsledků v první fázi testu, neznamenalo zlepšení i v hledání podobných mincí. Zde naopak v porovnání s předchozí verzí metody došlo k

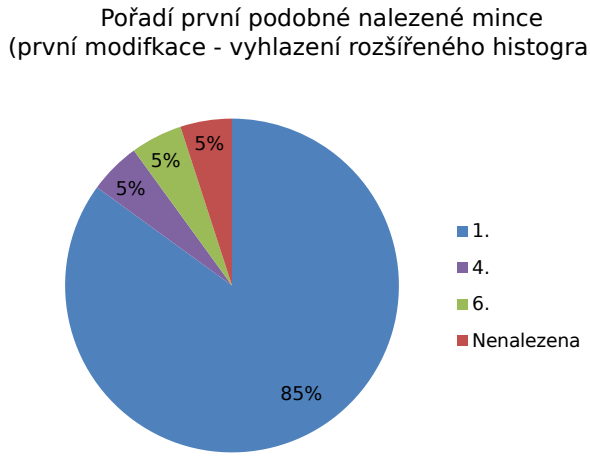



Obrázek 91: Výsledky vyhledávaní metody s vyhlazeným rozšířeným histogramem.

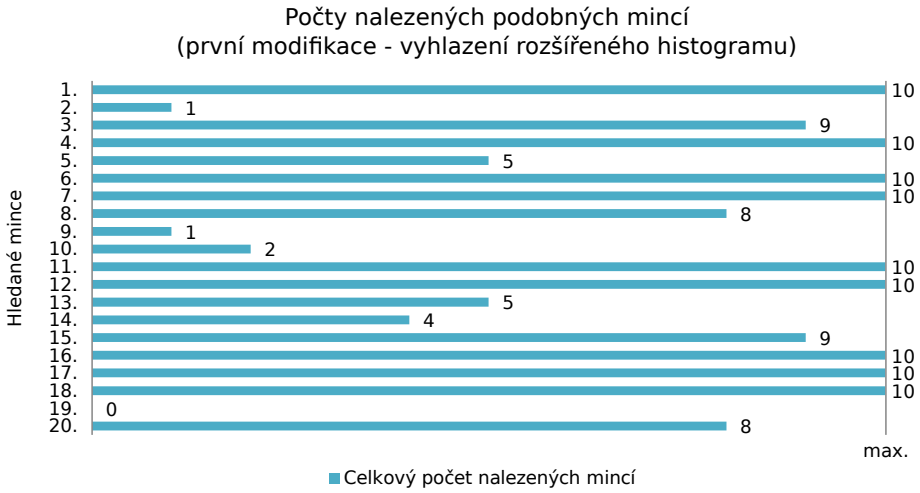
určitému zhoršení, jak co se týče přesnosti (graf na obr.93), tak celkového počtu podobných nalezených kusů (graf na obr. 93).

| | **Pořadí** | 1. | 2. | 3. | 4. | 5. | 6. | 7. | 8. | 9. | 10. | > |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Absolutní hodnoty** | **Shoda** | 64 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 |
| | **Alternativa** | 11 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 |
| | **Součet** | 75 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | – |
| | **Celkem** | 78 | | | | | | | | | | – |
| **[%]** | **Shoda** | 80 | 0 | 1.25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18.75 |
| | **Alternativa** | 13.75 | 1.25 | 0 | 0 | 0 | 1.25 | 0 | 0 | 0 | 0 | 2.5 |
| | **Součet** | 93.75 | 1.25 | 1.25 | 0 | 0 | 1.25 | 0 | 0 | 0 | 0 | – |
| | **Celkem** | 97.5 | | | | | | | | | | – |

Tabulka 3: Výsledky vyhledávání metody s vyhlazeným rozšířeným histogramem.




Obrázek 92: Procentuální rozložení pořadí, nalezení první podobné mince (test 3).




Obrázek 93: Celkové počty nalezených podobných mincí (test 3).

### 7.4.4 Zhodnocení první části úprav původní metody

První část modifikací měla za cíl odstranit faktické nedostatky předchozí metody, hlavně zajištění korektnosti dat na vstupech detektorů hran. Vcelku překvapením pak bylo zjištění, že oproti původní metodě se úspěšnost v první části testu snížila. Na druhou stranu v případě hledání podobných mincí se úspěšnost zvýšila, což je přesně ten cíl, ke kterému je směřováno. Vhledem k tomu, že na vstupu internetového katalogu mincí, se budou většinou objevovat obrázky, které nebyly použity pro předzpracování a je tak nutné nalézt podobný obrázek. V první fázi testu, kdy je hledána shoda by metoda měla, v ideálním případě, nalézt shodnou minci ve 100% případů. Proč tomu tak není, je dáno tím, že normování rotace mince nefunguje vždy stoprocentně správně.

Úprava metody normující natočení, zařazením bloku, který provede de facto druhé vyhlazení původního histogramu, napomohla ještě o něco málo zlepšit výpočet normujícího úhlu.

Jak je vidět z výsledků druhé části testů, kdy se počty nalezených podobných mincí pohybují mezi 140 a 150 kusy, což jsou téměř $3/4$ z 200 možných[58]. Dalo by se tak říci, že schopnost nacházet podobné mince se pohybuje někde mezi 70% až 75%.

## 7.5 Druhá část modifikací referenční metody

Druhá část modifikací se zaměřuje na zlepšení detekce hran, jelikož se jedná o stěžejní část celé metody. Je to dáno tím, že se od ní odvíjí, jak přesnost výpočtu normujícího úhlu, tak výsledný vektor, který je ukládán do databáze.

Cílem této modifikace je nalézt takové nastavení parametrů detektoru hran, aby nebylo detekováno tolik falešných hran, jako je tomu doposud. U většiny obrázků vypadají detekované hrany, po prahování, jako na obrázku 94. Jak je z něj vidět, ve výsledku se v obraze nachází mnoho malých hran, které vypadají spíše jako nežádoucí šum. Změnou několika parametrů, hlavně spodní a horní hodnotou prahu, se podařilo dosáhnout výstupů, které vypadají spíše jako na obrázku 95.

Obrázek 94: Hrany detekované s původním nastavením parametrů.

Obrázek 95: Hrany detekované s novým nastavením parametrů.

### 7.5.1 Nastavení metod

V prvním testu je zapojení jednotlivých metod stejné jako v diagramu na obrázku 86. Jediné v čem se liší je nastavení parametrů hranového detektoru. Pozměněné hodnoty jsou uvedeny v následujícím výpise, ty ostatní zůstávají stejné.

[58] V databázi se samozřejmě může nacházet i více kusů, bylo ale vždy vybráno pouze prvních deset nejpodobnějších kusů.

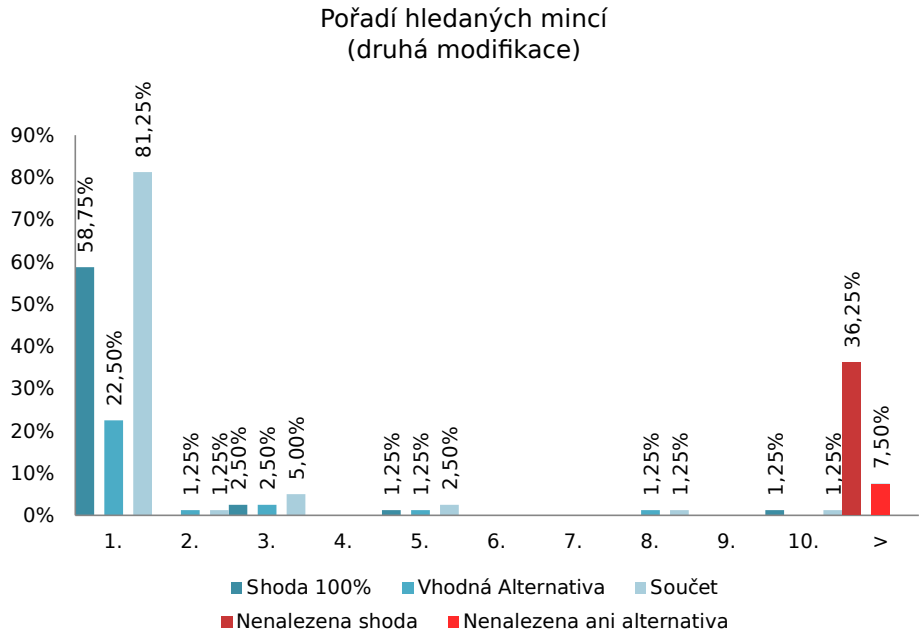
- Úhel = 0.6,
- spodní hodnota prahu: 15,
- horní hodnota prahu: 60.

Co se týká nastavení druhého testu, to zůstává stejné jako v druhém testu první modifikované verze. Pouze jsou stejným způsobem pozměněny parametry detektoru hran. Tyto dva testy mají za cíl zjistit, zda přesnější detekce hran pomůže zlepšit výsledky vyhledávání.

### 7.5.2 Výsledky vyhledávání metody s přesnější detekcí hran

Navzdory očekávání, lepší detekce hran nepřinesla žádné zlepšení, ba naopak jsou výsledky o poznání horší. Sice se v celkové úspěšnosti metoda pohybuje stále nad 90%, ale co se týče výsledků na jednotlivých pozicích, tak ty klesly. Jak je vidět z prvního grafu (obr. 96) v důsledků toho stoupla neúspěšnost nalezení shody na 36.25%, což je ve srovnání z předchozím testem, téměř 50% zhoršení.

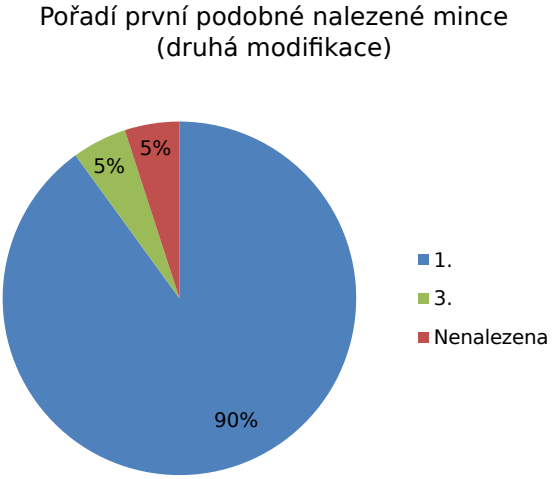
Navíc jak ukazuje třetí graf (obr. 98), klesla také schopnost hledání podobných mincí. Koláčový graf (obr. 97) sice ukazuje, že metoda je schopna v 90% případů nalézt podobnou mince na prvním místě, ale jejich celkový podíl klesl a to na 129 kusů. Oproti předchozím metodám se jedná o znatelný pokles, který není možné svést na statistickou chybu.



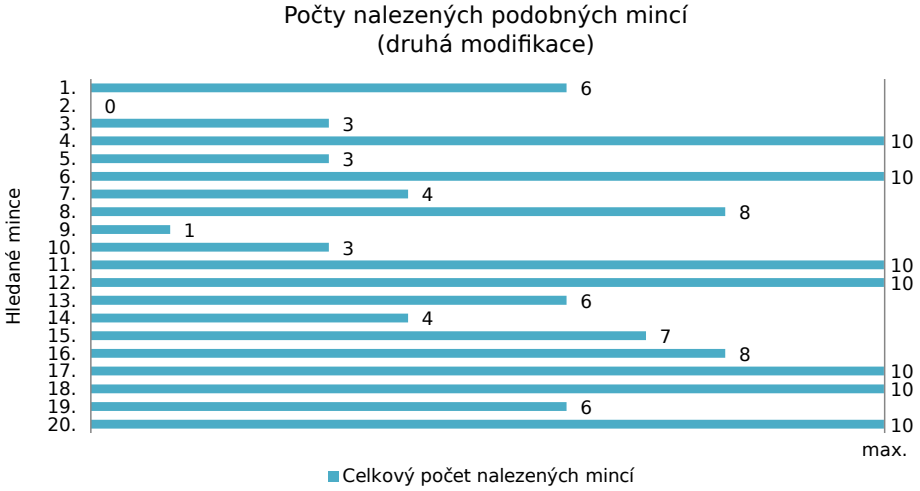

Obrázek 96: Výsledky vyhledávaní metody s přesnější detekcí hran.

| | Pořadí | 1. | 2. | 3. | 4. | 5. | 6. | 7. | 8. | 9. | 10. | > |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Absolutní hodnoty** | **Shoda** | 47 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 29 |
| | **Alternativa** | 18 | 1 | 2 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 6 |
| | **Součet** | 65 | 1 | 4 | 0 | 2 | 0 | 0 | 1 | 0 | 1 | – |
| | **Celkem** | 74 | | | | | | | | | | – |
| **[%]** | **Shoda** | 58.75 | 0 | 2.5 | 0 | 1.25 | 0 | 0 | 0 | 0 | 1.25 | 36.25 |
| | **Alternativa** | 22.5 | 1.25 | 2.5 | 0 | 1.25 | 0 | 0 | 1.25 | 0 | 0 | 7.5 |
| | **Součet** | 81.25 | 1.25 | 5 | 0 | 2.5 | 0 | 0 | 1.25 | 0 | 1.25 | – |
| | **Celkem** | 92.5 | | | | | | | | | | – |

Tabulka 4: Výsledky vyhledávání metody s přesnější detekcí hran.




Obrázek 97: Procentuální rozložení pořadí, nalezení první podobné mince (test 4).

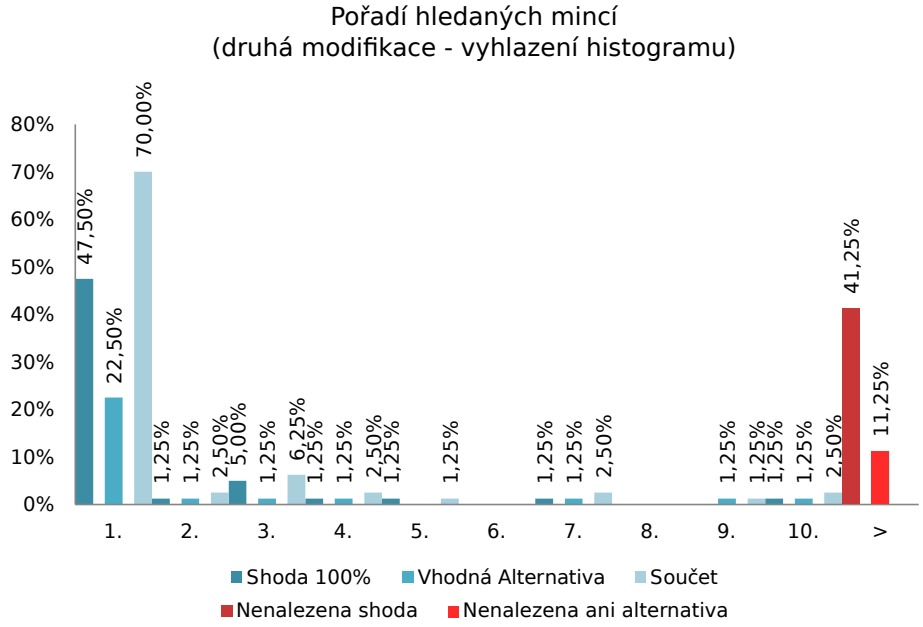



Obrázek 98: Celkové počty nalezených podobných mincí (test 4).

### 7.5.3 Výsledky vyhledávání metody s vyhlazenou verzí rozšířeného histogramu

Vyhlazení rozšířeného histogramu v tomto případě také nepomohlo a výsledky jsou ještě o něco horší. V tabulce 5 je vidět, že celková úspěšnost spadla pod 90% hranici a úspěšnost nalezení shodné mince na první pozicí je pouhých 47.5%.

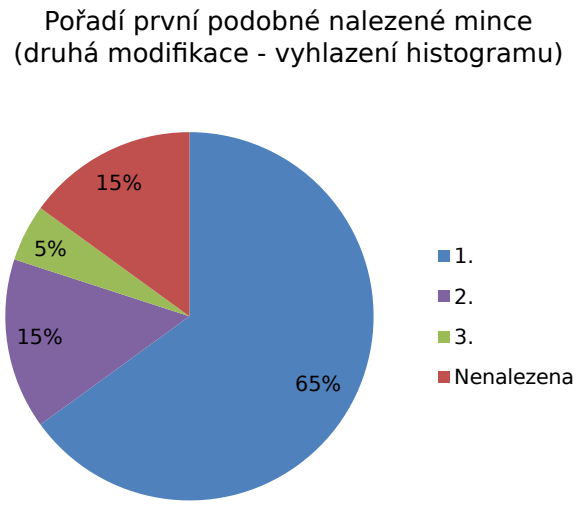
Také jak ukazují druhý a třetí graf (obr. 100 a 101) klesla schopnost rozpoznávat podobné mince. V případě tohoto testu, bylo nalezeno 110 podobných mincí z 200. Oproti nejlepší variantě je to propad o 39 nenalezených kusů.




Obrázek 99: Výsledky vyhledávaní druhé modifikované metody s vyhlazeným rozšířeným histogramem.

| | **Pořadí** | 1. | 2. | 3. | 4. | 5. | 6. | 7. | 8. | 9. | 10. | > |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Absolutní hodnoty** | **Shoda** | 38 | 1 | 4 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 33 |
| | **Alternativa** | 18 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 9 |
| | **Součet** | 56 | 2 | 5 | 2 | 1 | 0 | 2 | 0 | 1 | 1 | – |
| | **Celkem** | 71 | | | | | | | | | | – |
| **[%]** | **Shoda** | 47.5 | 1.25 | 5 | 1.25 | 1.25 | 0 | 1.25 | 0 | 0 | 1.25 | 41.25 |
| | **Alternativa** | 22.5 | 1.25 | 1.25 | 1.25 | 0 | 0 | 1.25 | 0 | 1.25 | 1.25 | 11.25 |
| | **Součet** | 70 | 2.5 | 6.25 | 1.25 | 1.25 | 0 | 2.5 | 0 | 1.25 | 2.5 | – |
| | **Celkem** | 88.75 | | | | | | | | | | – |

Tabulka 5: Výsledky vyhledávání druhé modifikované metody s vyhlazeným rozšířeným histogramem.




Obrázek 100: Procentuální rozložení pořadí, nalezení první podobné mince (test 5).


Obrázek 101: Celkové počty nalezených podobných mincí (test 5).

### 7.5.4 Zhodnocení druhé části úprav

Druhá část modifikace měla za cíl lépe detekovat hrany, protože se předpokládalo, že by to mohlo napomoci zlepšit výsledky vyhledávání. Jak bylo psáno výše, detekce hran je jedna ze základních operací, na které stojí celá metoda. Ovšem testy prokázaly, že snaha za každou cenu zlepšit výstupy jednotlivých operací nemusí vždy vést k lepším výsledkům.

Nyní vyvstává otázka, proč lépe detekované hrany na obrázku, mají tendenci výsledky spíše zhoršovat? Vysvětlením může být, že ona testovací množina je vlastně dost různorodá. Když by bylo například vybráno sto stejných mincí pořízených z různých zdrojů, tak každý z obrázků je svým způsobem jedinečný. Jedna mince může být v nejvyšší kvalitě, nafocená profesionálem, jiná může být zkorodovaná, další zase poškrábaná, atd.

Ona je tak vlastně určitá nepřesnost naopak žádoucí a ve výsledku to umožní nacházet v mnoha případech mince stejné a dokonce i mince s podobnými vzory.

## 7.6 Celkové srovnání metod normující úhel natočení mince

V této části jsou srovnány výsledky jednotlivých testovaných metod. Jak je vidět z grafu na obrázku 102, první tři varianty se pohybují zhruba na srovnatelné úrovni. Nejlépe z nich dopadla třetí varianta, u které se podařilo dosáhnout celkové úspěšnosti 97.5%, navíc schopnost nalézt minci hned na první pozicí se pohybuje na 80%. Na druhou stranu, z výsledků vynesených v druhého grafu (obr. 103), dopadla třetí metoda nejhůře z prvních třech. Mírně u ní klesla schopnost rozpoznat podobnou minci. Nicméně pokles není nikterak dramatický a vzhledem k výsledkům v první fázi testu, hodnotím třetí metodu jako nejvhodnější pro další nasazení v rámci internetového katalogu mincí[59].

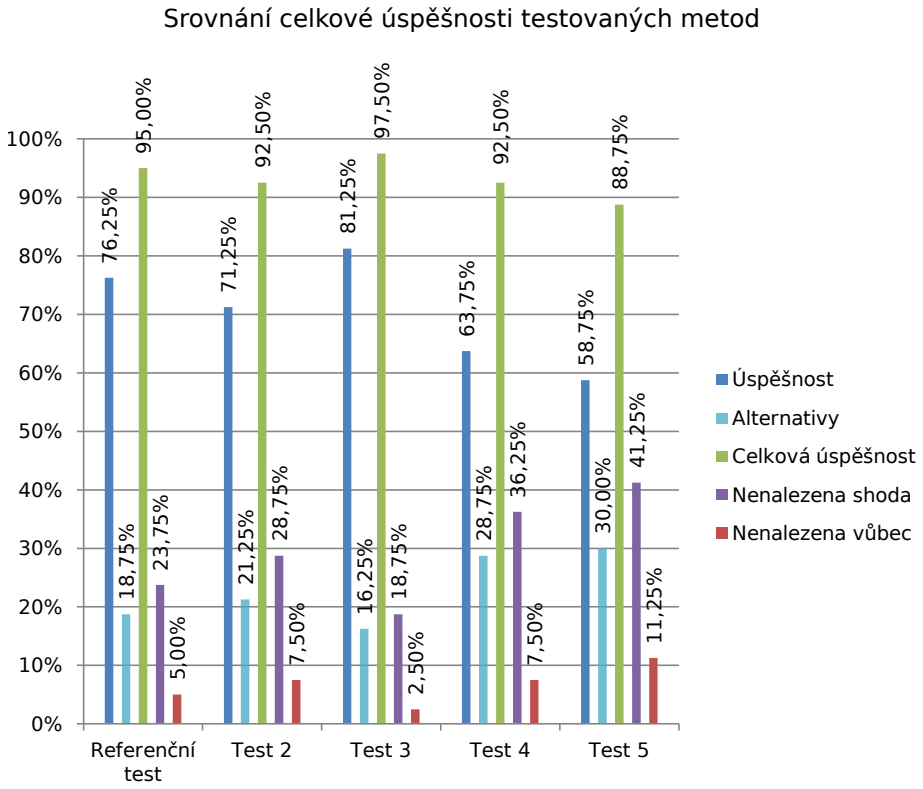
Testy s přesnější detekci hran naopak ukázaly, jak je důležité zachovat v obraze po jejich detekci určitý šum. Ten je následně klíčem k tomu, že je metoda jako celek robustní a zvládá rozeznávat i mince, na kterých se mohou nacházet různé vady. Nakonec ani tak nevadí, že metoda normování rotace není stoprocentní, jelikož při dostatečném počtu mincí se utvoří skupinky sobě podobných mincí s podobným natočením a v případě, že se minci na vstupu nepodaří „správně natočit“, může se najít v databázi jiná skupina s daným natočením a mince je tak téměř vždy nalezena.

Z výsledků prvních třech testů vyplývá ještě jedna věc a to, že hledat minci na prvních deseti pozicích je možná až zbytečné moc. S dosaženými výsledky by v katalogu mohly být vypisovány klidně první tři nebo pět nejpodobnějších mincí. V druhém případě by ani nemusela klesnout naměřená přesnost.

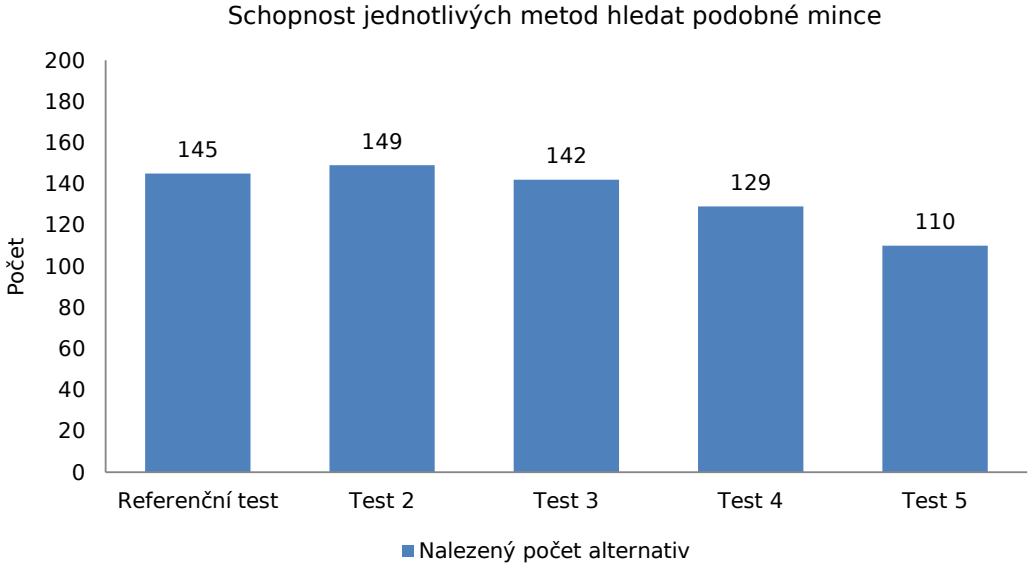
| | Referenční test | | Test 2 | | Test 3 | | Test 4 | | Test 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Počet | [%] | Počet | [%] | Počet | [%] | Počet | [%] | Počet | [%] |
| Úspěšnost | 61 | 76.25 | 57 | 71.25 | 65 | 81.25 | 51 | 63.75 | 47 | 58.75 |
| Alternativy | 15 | 18.75 | 17 | 21.25 | 13 | 16.25 | 23 | 28.25 | 24 | 30 |
| Celk. úspěšnost | 76 | 95 | 74 | 92.5 | 78 | 97.5 | 74 | 92.5 | 71 | 88.75 |
| Nenalezená shoda | 19 | 23.75 | 23 | 28.75 | 15 | 18.75 | 29 | 36.25 | 33 | 41.25 |
| Nenalezena vůbec | 4 | 5 | 6 | 7.5 | 2 | 2.5 | 6 | 7.5 | 9 | 11.25 |

Tabulka 6: Srovnání úspěšnosti jednotlivých metod.

[59] `http://identifycoin.vsb.cz`




Obrázek 102: Srovnání úspěšnosti jednotlivých metod.




Obrázek 103: Srovnání úspěšnosti jednotlivých metod v hledání podobných mincí.

## 7.7 Metoda popisující obraz nezávisle na rotaci

V rámci projektu na rozpoznávání pamětních a oběžných mincí byly testovány i jiné přístupy popisu obrazu tak, aby nemusel být brán zřetel na jeho natočení. Poprvé se o to pokoušel již Ing. Petr Kašpar ve své bakalářské prací a dané metodě se také zmiňuje v [3]-kap.4.1. Její princip byl vcelku velmi jednoduchý. Fungovala tak, že minci jakoby roztočila vysokou rychlostí, čímž vznikly na obrázku soustředné prstence. Následně se od středu mince k jejímu okraji provedl řez a hodnoty v tomto řezu se použily jako vektor pro další porovnávání. Metoda měla řadu nevýhod a nefungovala nijak zvlášť dobře.

Cílem metody popisované v této kapitole, je vytvořit něco jako otisk mince, který bude nezávislý na jejím natočení. Opět je vycházeno z obrazu po polární transformaci, který eliminuje problém rotace na problém posunutí. V tuto chvíli, kdyby se vytvořila sada obrazů, které by se posouvaly pouze v horizontálním směru, přeložily přes sebe a jejich hodnoty zprůměrovaly, vzniklo by něco velmi podobného jako v předchozím případě. V metodě popisované zde se „obrazy“ posouvají v obou směrech.

### 7.7.1 Princip metody

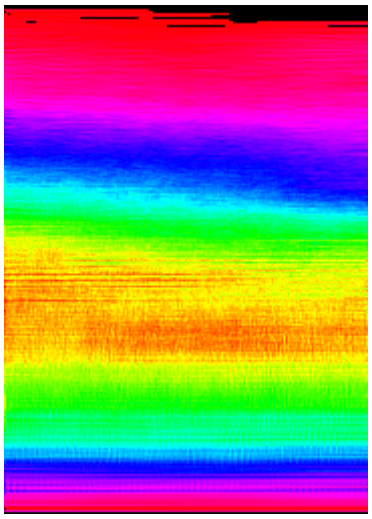
Průměrování barev nebo jasů jednotlivých pixelů není zrovna nejvhodnější způsob. Proto před samotným vytvářením otisku mince je provedeno několik stejných kroků jak v předchozích metodách. Vstupem metody je vlastně stejný obraz jako v případě metody počítající normující úhel. Vstupem metody tak je binární obraz po polární transformaci s detekovanými hranami.

Na vstupní obraz se nyní pohlíží, jako na jakousi šablonu s otvory. Každý hranový (bílý) pixel tvoří jeden otvor. V prvním kroku se šablona přiloží na prázdnou matici a do buněk pod jednotlivými otvory se přičte jednička. V druhém kroku je maska na matici přiložena tak, aby lícovala první označená buňka s druhým otvorem na masce a opět se inkrementují hodnoty na pozicích pod otvory. V třetím kroku je k první označené buňce přiložena maska tak, aby lícoval třetí otvor. Takto se pokračuje, dokud nejsou vyčerpány všechny otvory na masce. V tu chvíli se v původním obraze získaném v prvním kroku vybere druhá označená buňka a celý proces se zase opakuje. Pokračuje se tak dlouho, dokud nejsou z prvního obrazu vybrány všechny označené buňky. Dojde tak k tomu, že je slícována každá buňka s každým otvorem, čímž vznikne vcelku zajímavá struktura, která je odolná vůči natočení původního obrazu. Jak vypadá vizualizace podobné struktury ukazuje obrázek 105, čím je barva červenější, tím je hodnota v matici vyšší. Vizualizovaná struktura patří k minci na obrázku 104 a byla vygenerována trochu odlišnou metodou, než jak byla na úvod pro představu popsána.

Metoda není opět zcela stoprocentní, zde hodně záleží na tom jak jsou hrany ve výsledku detekovány. Stává se, že pro různá natočení obrazu jsou hrany detekovány trochu odlišně, což způsobí že výsledné struktury nejsou zcela totožné, nicméně jsou si hodně podobné. Několik ukázek se nachází v příloze F, kde jsou vidět výsledky, jak pro různá natočení stejné mince, tak i výsledky pro jiné mince.

Pseudokód (alg. 4) ukazuje jak probíhá výpočet otisku mince. Rozdíl v použité verzi, oproti té popisované je, že na řádku 24. ve výpisu se *y-psilonová* souřadnice přičítá. V po-


Obrázek 104: Originální obrázek.



Obrázek 105: Popis (otisk) mince.

pisu nahoře je napsáno, že se šablona postupně posouvá. To by znamenalo, že obě souřadnice by se musely odečíst od aktuálně označeného středu (*marked_position*). Nicméně bylo zjištěno, že když se *x-ové* souřadnice odečtou a *y-psilonové* sečtou, je výsledná struktura horizontálně symetrická podle středu. Tak je možné ukládat pouze polovinu výsledku, bez ztráty informace. Z tohoto důvod je nová šířka (*new_width*) určena, jako polovina šířky obrazu daného na vstupu. Nová výška (*new_height*) je dvakrát větší, protože se obrázky při překrývání mohou dostat maximálně na dvojnásobek původní výšky. Výsledná matice je tak rozměru $new\_width \times new\_height$.

V pseudokódu se již nenachází část s relativizací hodnot ve výsledné matici. Díky velkému množství překrytí, ke kterým dojde během výpočtu, jsou v matici na konci dost velká čísla. Proto se všechna převádí do rozsahu 0 až 1000. Navíc při ukládání výsledku do databáze je matice opět převedena na vektor a opět se využívá masky (obdélníkové masky), která zredukuje velikost koncového vektoru. Hodnoty v rámci jednoho bloku jsou sečteny a nebylo tak vhodné mít zde příliš velká čísla. Při vyhledávání jsou výsledné vektory porovnávány stejně, jako v předchozích metodách.

Složitost této metody, je $O(n^2)$, kde $n$ je počet hranových pixelů v obraze. Nejhorší případ, který může nastat je, že bude na vstup poslán obraz bílé barvy[60]. V takovém případě se provede $(M \times N)^2$ operací, kde $M$ je šířka vstupního obrázku a $N$ jeho výška. Toto již není moc použitelné, avšak u praktických testů byla průměrná rychlost metody asi $3\times$ pomalejší než u předchozích verzí. Doba předzpracování metody popsané v 7.4.3, byla zhruba necelé tři a půl hodiny. Předzpracování pětadvaceti tisícové kolekce, této metodě trvala něco málo přes devět hodin.

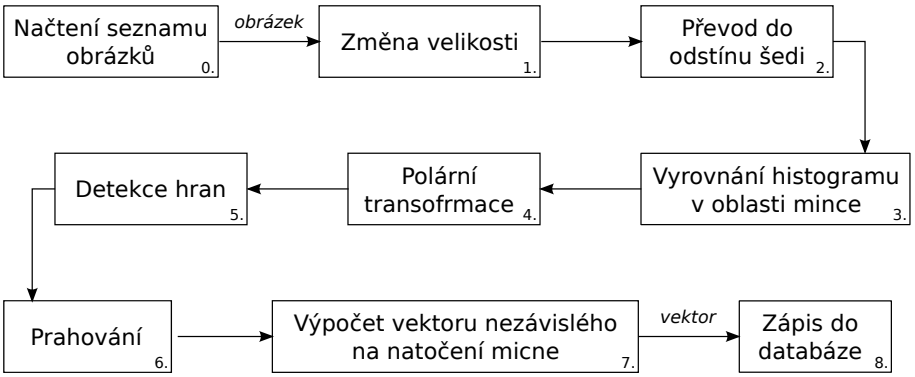
[60] Pixely hran jsou, v aplikaci a v typu `BinaryImage`, označený bílo barvou.

**Algoritmus 4** Výpočet otisku popisujícího minci nezávisle na rotaci.

```
1:  function COMPUTE_COIN_DESCRIPTION(binary_image, width, height)
2:      if width mod 2 = 0 then
3:          new_width ← width/2 + 1
4:      else
5:          new_width ← width/2
6:      end if
7:      new_height ← 2 * height
8:      for x ← 0, width do                    ▷ Načte souřadnice hranových pixelů.
9:          for y ← 0, height do
10:             if binary_image[x][y] = WHITE then
11:                 coordinates ← (x, y)
12:             end if
13:         end for
14:     end for
15:     for i ← 0, coordinates.size do
16:         marked_position ← coordintes[i]
17:         for j ← 0, coordinates.size do
18:             current_position ← coordinates[j]
19:             x ← current_position.x − marked_position.x
20:             if x < 0 then
21:                 x ← width + x              ▷ Začátek a konec x-ové osy jsou spojeny.
22:             end if
23:             if x < width2 then
24:                 y ← current_position.y + marked_position.y
25:                 description[x][y] ← desription[x][y] + 1
26:             end if
27:         end for
28:     end for
29:     return description
30: end function
```

### 7.7.2 Nastavení metody

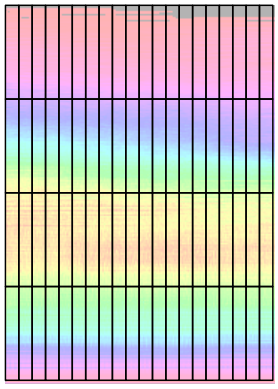
Na obrázku 106 je vidět kompletní zapojení jednotlivých operací do výsledné metody. Natavení operací




Obrázek 106: Kompletní zapojení metody popisující obraz mince nezávisle na natočení.

Nastavení operací 1 až šest je stejné jako v první modifikaci (kap. 7.4). U detektoru hran jsou tak nastaveny parametry, aby bylo detekováno větší množství hran v obraze. Byla zkoušena testována i varianta s přesnější detekcí, v takovém případě byla rychlost zpracování již srovnatelná s předchozími metodami, ale výsledky nebyly příliš dobré. V následující části jsou tak uvedeny pouze výsledky pouze této jedné varianty.

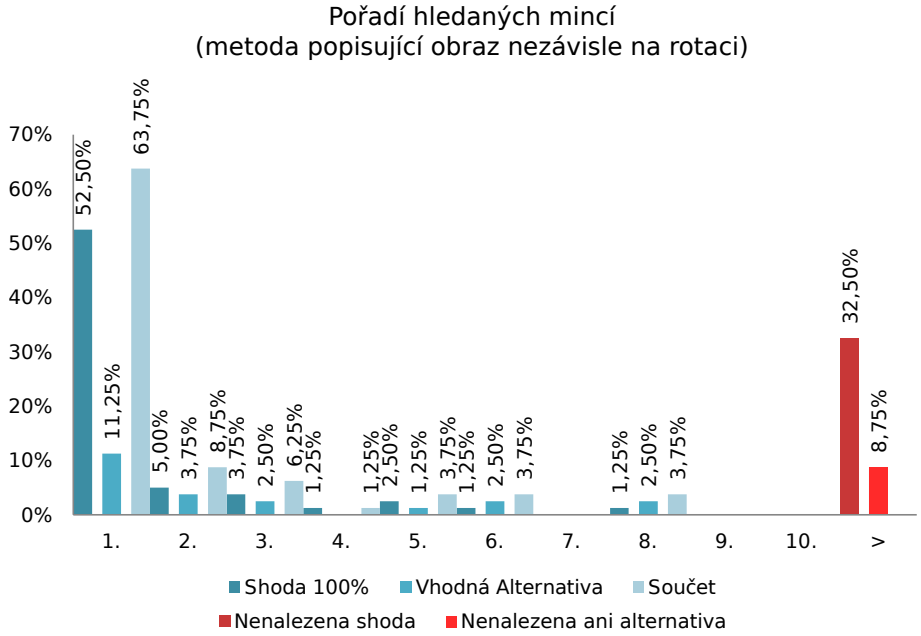
Sedmá operace v pořadí (*výpočet vektoru nezávislého na rotací*) se skládá ze dvou kroků. V prvním je proveden výpočet struktury popisující obraz. V druhém kroku je na výsledek aplikována čtvercová maska, pomocí které je vytvořen výstupní vektor, jenž je přímo uložen do databáze nebo je podle něj v ní vyhledáváno. Operace v aplikaci má tak krom vstupního obrázku, také dva vnitřní vstupní parametry a to šířku a výšku jedné oblasti v masce. Ty jsou nastaveny na 10px šířku a 63 px výšku. Výsledná matice má rozměry při daném nastavení $254 \times 181$px. Aplikací masky na obraz má výsledný vektor 72 položek.



Obrázek 107: Aplikace masky na výslednou strukturu.

### 7.7.3 Dosažené výsledky

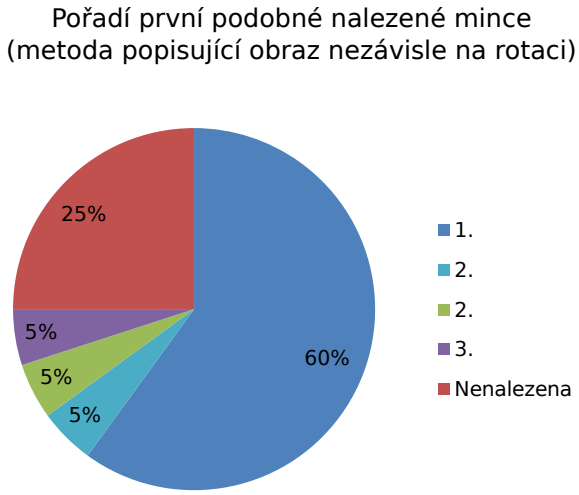
Výsledky dosažené za pomocí této metody, v první části testu, dopadly celkem slušně, jak ukazuje tabulka 7. Dokonce celková úspěšnost je 91.25%. Avšak z prvního grafu (obr. 108) je vidět, že dílčí výsledky jsou, oproti předchozím metodám, více rozprostřeny na různých pozicích. Z druhé části testu, kdy byly hledány podobné mince, pak úspěšnost této metody rapidně klesla dolů na pouhých 49 podobných nalezených kusů. Druhý graf (obr. 109) ukazuje, že metoda byla úspěšná jen z 60% v nalezení podobné mince na první pozici, což ostatně potvrzuje i první graf, kde je vidět, že po sečtení shodných a alternativních mincí, na první pozici, je necelých 64%. Na posledním grafu (obr. 110) je vidět, že pouze v jediném případě se podařilo nalézt na prvních deseti pozicích podobné mince, zbytek se pohybuje spíše od pěti kusů níže.




Obrázek 108: Výsledky vyhledávaní metody popisující obraz nezávisle na rotaci.

| | Pořadí | 1. | 2. | 3. | 4. | 5. | 6. | 7. | 8. | 9. | 10. | > |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Absolutní hodnoty** | **Shoda** | 42 | 4 | 3 | 1 | 2 | 1 | 0 | 1 | 0 | 0 | 26 |
| | **Alternativa** | 9 | 3 | 2 | 0 | 1 | 2 | 0 | 2 | 0 | 0 | 7 |
| | **Součet** | 51 | 7 | 5 | 1 | 3 | 3 | 0 | 3 | 0 | 0 | – |
| | **Celkem** | 73 | | | | | | | | | | – |
| **[%]** | **Shoda** | 52.5 | 5 | 3.75 | 1.25 | 2.5 | 1.25 | 0 | 1.25 | 0 | 0 | 32.5 |
| | **Alternativa** | 11.25 | 3.75 | 2.5 | 0 | 1.25 | 2.5 | 0 | 2.5 | 0 | 0 | 8.75 |
| | **Součet** | 63.75 | 8.75 | 6.25 | 1.25 | 3.75 | 3.75 | 0 | 3.75 | 0 | 0 | – |
| | **Celkem** | 91.25 | | | | | | | | | | – |

Tabulka 7: Výsledky vyhledávaní metody popisující obraz nezávisle na rotaci.




Obrázek 109: Procentuální rozložení pořadí, nalezení první podobné mince (test 6).


Obrázek 110: Celkové počty nalezených podobných mincí (test 6).

### 7.7.4 Zhodnocení metody

Tato metoda byla původně navržena jako alternativa pro jednotný popis stejného obrazu mince, který bude nezávislý na jeho natočení. To se i vcelku podařilo, sice není metoda stoprocentní, ale jak ukázaly testy s různě natočenými mincemi, stavět by se na ní dalo. Nicméně po provedení testů na reálných datech, kdy mohou být mince, ne jen různě natočeny, ale také se mohou nacházet v různě kvalitním stavu, atp. se nakonec metoda příliš neosvědčila. Hlavně z pohledu hledání podobných mincí.

Na druhou stranu v porovnání s podobnými přístupy, které byly zkoušeny v minulosti ([3] – kap.4.1) se jedná o variantu s nejlepšími výsledky, kdy se podařilo zlepšit, jak přesnost, tak snížit výpočetní složitost a není zcela vyloučeno její použití v katalogu,

jako tomu bylo v předchozích případech. Ovšem ve srovnání s výsledky dosažených v rámci prvních modifikací původní metody, jsou výsledky této metody velmi slabé, což je hlavní argument proč byla pro další použití doporučena metoda založená na normování natočení obrazu mince.

# 8 Závěr

Výsledkem této práce je nástroj, sloužící pro modelování a testování komplexních algoritmů, složených z více či méně složitých bloků. Nástroj si kladl za cíl hlavně zjednodušit vytváření nových metod a urychlit jejich testování. Toto mohu z vlastní zkušenosti potvrdit, jelikož na závěr proběhla celá řada různých testů, s různými dílčími modifikacemi. Vytvoření nové metody nebo úprava té stávající, tak byla otázka několika málo minut. Pokud bylo třeba dodat novou operaci do aplikace, čas se prodloužil v závislosti na složitosti dané metody. Čas potřebný na zakomponování nové operace do aplikace je minimální a po programátorovi se nechce v kódu o moc více než by musel stejně napsat. Vytvořené a uložené metody je také možné spouštět i v prostředí bez grafického rozhraní, např. na serveru.

V druhé polovině práce se již nachází popis metod použitých pro předzpracování fotografií mincí tak, aby mohly být uloženy v databázi, s možností následného vyhledávání. Ze začátku bylo provedeno srovnání mezi novou a původní verzí, kterou použil v katalogu Ing. Petr Kašpar. Testy se zaměřovaly na porovnání rychlosti obou verzí a toho zda obě verze poskytují stejné výstupy pro stejná vstupní data. Byly také objeveny dvě chyby v předchozí aplikaci. První, která pouze počítala navíc něco co dále nebylo použito a tím pádem zpomalovala výpočet. Druhá chyba, ale již měla vliv na výsledky vyhledávání. Po jejím odstranění stoupla úspěšnost vyhledávání rapidně nahoru. Co se týká výsledků srovnání rychlostí nová aplikace je o něco rychlejší, jelikož nyní, pokud je to možné, spouští operace paralelně.

Krom opravy chyby, byly navrženy ještě další modifikace metody. Jedny měli za cíl odstranit jistou nekonzistenci v původní metodě, druhé pak ověřit domněnku, že lépe detekované hrany budou mít za následek, lepší výsledky ve vyhledávání. Tato domněnka se ukázala být mylná a oproti očekávání vedla k tomu, že úspěšnost vyhledávání klesala dolů. Ukázalo se tak, že pro reálná data je naopak lepší jistá míra nepřesnosti, která zaručí, že metoda bude schopna nalézt i podobné obrázky mincí. Nejlépe dopadla první modifikovaná metoda s vyhlazením rozšířeného histogramu za pomocí Gaussovy funkce, jejíž celková úspěšnost byla 97.5%. U testu vyhledávání podobných mincí, jejichž přesné vzory nebyly do databáze uloženy, pak ze všech vzorů, které byly pro danou sadu vybrány, téměř tři čtvrtiny označeny správně.

Na závěr byla představena ještě metoda, která je založena na jiném přístupu. Nesnaží se totiž u obrazů mincí normovat jejich natočení, ale vytvořit takový popis, který bude invariantní vůči rotaci. Z pohledu popisu obrazu, který je nezávislý na rotaci, je metoda vcelku úspěšná, i když ne stoprocentní. Z pohledu úspěšnosti ve vyhledávání podobných mincí, se ale nemůže srovnávat s výsledky dosaženými pomocí metod využívající normování natočení.

Během práce bylo také publikováno několik odborných článků popisujících řešenou problematiku. Ty byly přijaty, jak na českých ([20], [21], [22], [4], [23]), tak mezinárodních([24], [25], [26]) konferencích a vydány v příslušných sbornících.

## 8.1 Návrhy na navazující práci

V práci je stále na co navazovat, je možné pokračovat ve vylepšování vyhledávacího algoritmu. V tom to případě se jeví jako zajímavé využít některých z metod, které jsou schopny v obraze detekovat významné body nezávisle na natočení obrazu. Další cestou, jak by bylo možné zpřesňovat výsledky vyhledávání, může být využití zpětné vazby, kterou mohou poskytnout uživatelé internetového katalogu. Tímto způsobem tak do katalogu zakomponovat jistou formu učení se z chyb.

Krom zlepšování výsledků vyhledávání, je možné navázat na zde implementovaný nástroj a např. pro něj připravit sadu dalších operací, popř. jej ještě více zobecnit a využít pro zcela jinou problematiku než je zpracování obrazu.

Co v celém konceptu stále chybí a mohlo by být zajímavé, je aplikace pro chytré telefony s fotoaparátem. Ta má největší potenciál, jak zaujmout velké množství lidí tak, aby začali katalog aktivně využívat. Přeci jen, kdo v dnešní době bude fotit mince a nahrávat obrázky na web, když by k tomu mohl jednoduše využit svůj mobilní telefon.

# 9 Literatura


[1] Petr Kašpar. *Internetový katalog pro rozpoznávání oběžných a pamětních mincí*. Bakalářská práce, Katedra informatiky, VŠB – TU Ostrava, 2008.

[2] Martin Šurkovský. *Praktické řešení porovnávání rastrového obrazu mincí nezávisle na rotaci.* Bakalářská práce, Katedra informatiky, VŠB – TU Ostrava, 2009.

[3] Petr Kašpar. *Předzpracování, indexace a vyhledávání v rozsáhlé kolekci obrazových dat*. Diplomová práce, Katedra informatiky, VŠB – TU Ostrava, 2011.

[4] Martin Šurkovský, Radoslav Fasuga, and Petr Kašpar. Aplikace k hromadnému předzpracování obrazových dat pro účely vylhledávání. In *Sborník 30. konference o geometrii a grafice*, pages 231–328, Praha, Czech Republic, 2010. Ed. Miroslav Lávička et al., matfyzpress.

[5] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[6] James C. Browne, Jack Dongarra, Syed I. Hyder, Keith Moore, and Peter Newton. Visual programming and parallel computing, 1994.

[7] G. Kahn. The semantics of a simple language for parallel programming. *Proc. IFIP Congress*, pages 471–475, 1974.

[8] W.B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, February 1982.

[9] Visual programming language. The free dictionary. [online - 19.3.2012] `http://encyclopedia2.thefreedictionary.com/visual+programming+language`.

[10] Margaret M. Burnett and Marla J. Baker. A classification system for visual programming languages. Technical report, Oregon State University, Corvallis, OR, USA, 1993.

[11] Stručný popis nástroje labview. Wikipedia. [online - 12.3.2012] `http://en.wikipedia.org/wiki/LabVIEW`.

[12] Sanner M.F., Stoffler D., and Olson A.J. Viper, a visual programming environment for python. In *Proceedings of thre 10th International Python conference*, pages 103 – 115, February 2002.

[13] Paradal C., Dufour-Kowalski S., Boudon F., and Fournier C. Goding C. Openalea: a virtual programming and component-based software platform for plant modelling. *Functional Plant Biology*, 35:751–760, 2008.

[14] S. Böhm and M. Běhálek. Kaira: Modelling and generation tool based on petri nets for parallel applications. In *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference*, pages 403–408, April 2011.

[15] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, 1st edition, 2009.

[16] M. Běhálek, S. Böhm, P. Krömer, M. Šurkovský, and O. Meca. Parallelization of ant colony optimization algorithm using kaira. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference*, pages 510 –515, November 2011.

[17] Deutsh limit. Wikipedia. [online - 19.3.2012] `http://en.wikipedia.org/wiki/Deutsch_limit`.

[18] Rudolf Pecínovský. *Návrhoé vzory*. Computer Press, a.s., 2007.

[19] John Zukowski. *The Definitive Guide to Java Swing*. Apress, 3rd edition, 2005.

[20] Radoslav Fasuga, Petr Kašpar, and Martin Šurkovský. Rozpoznávání obrazu a indentifikace pamětních mincí. In *Sborník 29. konference o geometrii a grafice*, pages 113–120, Praha, Czech Republic, 2009. Pavel Pech et al.

[21] Martin Šurkovský, Radoslav Fasuga, and Petr Kašpar. Možnosti normování rotace rastrového obrazu mincí a popis hashovacími funkcemi nezávislými na rotaci. In *Sborník 29. konference o geometrii a grafice*, pages 289–298, Praha, Czech Republic, 2009. Pavel Pech et al.

[22] Petr Kašpar, Radoslav Fasuga, and Martin Šurkovský. Využití specifických oblastí v obraze při vyhledávání. In *Sborník 29. konference o geometrii a grafice*, pages 173–180, Praha, Czech Republic, 2009. Pavel Pech et al.

[23] Martin Šurkovský and Radoslav Fasuga. Despr - nástroj pro snadné skládání algoritmů. In *Sborník 31. konference o geometrii a grafice*, pages 267–274, Ostrava, Czech Republic, 2011. Lavička, Miroslav and Němec, Martin.

[24] Radoslav Fasuga, Petr Kašpar, and Martin Šurkovský. Design of searchable commemorative coins image library. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ISVC '09, pages 397–406, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Radoslav Fasuga, Martin Šurkovský, and Petr Kašpar. Utilization of an image specific areas in searching. In *ICDS '10: Proceedings of the 2010 Fourth International Conference on Digital Society*, pages 279–284, Washington, DC, USA, 2010. IEEE Computer Society.

[26] Radoslav Fasuga, Martin Šurkovský, Petr Kašpar, and Martin Němec. Construction and interpretation of data structures for rotating images of coins. In Vaclav Snasel, Jan Platos, and Eyas El-Qawasmeh, editors, *Digital Information Processing and Communications*, volume 188 of *Communications in Computer and Information Science*, pages 439–447. Springer Berlin Heidelberg, 2011.

# A Obsah CD

**API/** zdrojové soubory API implementované aplikace, javadoc dokumentace, buildovací skripty a samotný vytvořený jar balík.

**Despr_1.0/** v tomto adresáři se nachází vše potřebné a je možné aplikaci z něj spouštět. Aplikace je připravená a nachystaná se všemi rozšířeními. Navíc se zde nachází buildovací skripty, zdrojové soubory a javadoc dokumentace.

**installed_extensions/** adresář obsahuje archivy použitých rozšíření v aplikaci. V archivu se nachází jak zkompilovaná verze, tak zdrojové soubory.

V archivu `Despr_DatabaseOperations.zip` jsou uloženy i kódy s metodami pro rozšíření MySQL databáze, adresář `c_src`.

**used_libraries/** adresář obsahuje knihovny, které byly použity při překladu aplikace nebo některých rozšíření.

**saved_tests_methods/** adresář obsahuje uložené soubory představující namodelované metody použité při testování. Povětšinou jsou uvedeny jenom vyhledávací verze. Algoritmy sloužící pro předzpracování se liší v ukončující operaci. Testovací metody se zde nachází spíše kvůli nastavení hodnot vnitřních parametrů operací. A ty jsou stejné, jak u metod které data zpracují a uloží do databáze, tak u metod které se je pokouší následně nalézt. Číslování testovacích metod koresponduje s číslováním v textu, resp. v pořadí v jakém byly jednotlivé metody a výsledky testů představeny.

# B Uživatelská příručka

Zde je ve stručnosti popsáno, jak pracovat s aplikací. Je řečeno, jak se aplikace spouští a co vyžaduje proto to, aby mohla být spuštěna. Jsou zde také popsány funkce prostředí. Nástroj jako takový by měl být pro uživatele docela snadno pochopitelný.

## B.1 Spuštění aplikace

Aplikace se spouští ze spustitelného `jar` balíku (*Despr.jar*) v adresáři, ve kterém je uložena. Není možné balík přesunout do jiné složky, alespoň ne samostatně. Ke svému běhu totiž potřebuje následující adresáře:

- `extensions`,
- `lib`,
- `log`,
- `resources`.

V prvním jsou uloženy knihovny, které aplikace načítá za běhu. Jedná se o balíky rozšiřujících operací, typů a typových rozšíření. V adresáři `lib` se nachází externí knihovny, které byly přidány při kompilaci nástroje. Nachází se zde např. API aplikace nebo ovladač pro připojení k MySQL databázi. V adresáři `log`, jak název napovídá, lze nalézt logovací soubory, do kterých aplikace vypisuje chyby, které nastaly při jejím běhu. Nakonec adresář `resource` obsahuje veškeré externí zdroje, jež nástroj využívá. Nachází se zde ikony, lokalizační soubory a informace o nastavení aplikace.

### B.1.1 Verze bez grafického rozhraní

Verze bez grafického rozhraní slouží hlavně pro spouštění již vytvořených algoritmů. Slouží pro případy, kdy má uživatel připravené nějaké úlohy, které bych chtěl nechat zpracovat, ale nemá na daném počítači nainstalováno GUI, typicky servery. Zde je možné využit této možnosti a připravenou úlohu pouze spustit. Aplikace se spouští příkazem:

```
java -jar Despr.jar cesta/k/souboru.despr.zip
```

Po spuštění je načten graf, vypíše se v textové podobě jeho nastavení na standardní výstup a zeptá se zda má být spuštěn či nikoliv. Zmáčknutím enteru nebo napsání potvrzující hlášky[61], je graf spuštěn. Chvíli může trvat, než se objeví informace o stavu zpracování, protože aplikace na základě prvního běhu začíná odhadovat čas a vypisovat stav, jak daleko se ve výpočtu nachází. Pokud je algoritmus časově náročný nebo je načítáno velké množství dat, muže to trvat i několik minut. Ukázka spuštěné aplikace v prostředí bez GUI je vidět na obrázku 111.

[61] V tomto případě záleží na lokalizaci.

```
sur096@Martin-Linux:~/Desktop/Despr_1.0$ java -jar Despr.jar ~/zpracovani_velkych_obrazku.despr.zip

Welcome to the application: Desrp v1.0
*****************************************************************************************************************
* You loaded this graph:                                                                                        *
*---------------------------------------------------------------------------------------------------------------*

Load images[0]@1
  Source directory:cz.vsb.cs.sur096.despr.types.Directory = /home/sur096/Dropbox/Camera Uploads
  Recursive browse:java.lang.Boolean = false
  Random browse:java.lang.Boolean = false

Save image[3]@2
  Destination directory:cz.vsb.cs.sur096.despr.types.Directory = /home/sur096/C_TEST_LARGE_IMAGE
  Subdirectory path:java.lang.String =
  Name specification:java.lang.String =

File extractor[1]@3

Gray-scale[1]@5

Histogram equalize[2]@7


LoadImages            @1(Image file)    -->     FileExtractor         @3(File)
LoadImages            @1(Image)         -->     Grayscale             @5(Input image)
Grayscale             @5(Output Image)  -->     HistogramEqualize     @7(Input image)
HistogramEqualize     @7(Output image)  -->     SaveImage             @2(Input image)
FileExtractor         @3(File name)     -->     SaveImage             @2(File name)

*---------------------------------------------------------------------------------------------------------------*
Do you want to run this graph? [YES/no]
yes
|======>                                          |   6%   [Time left: 00:20:27]
```

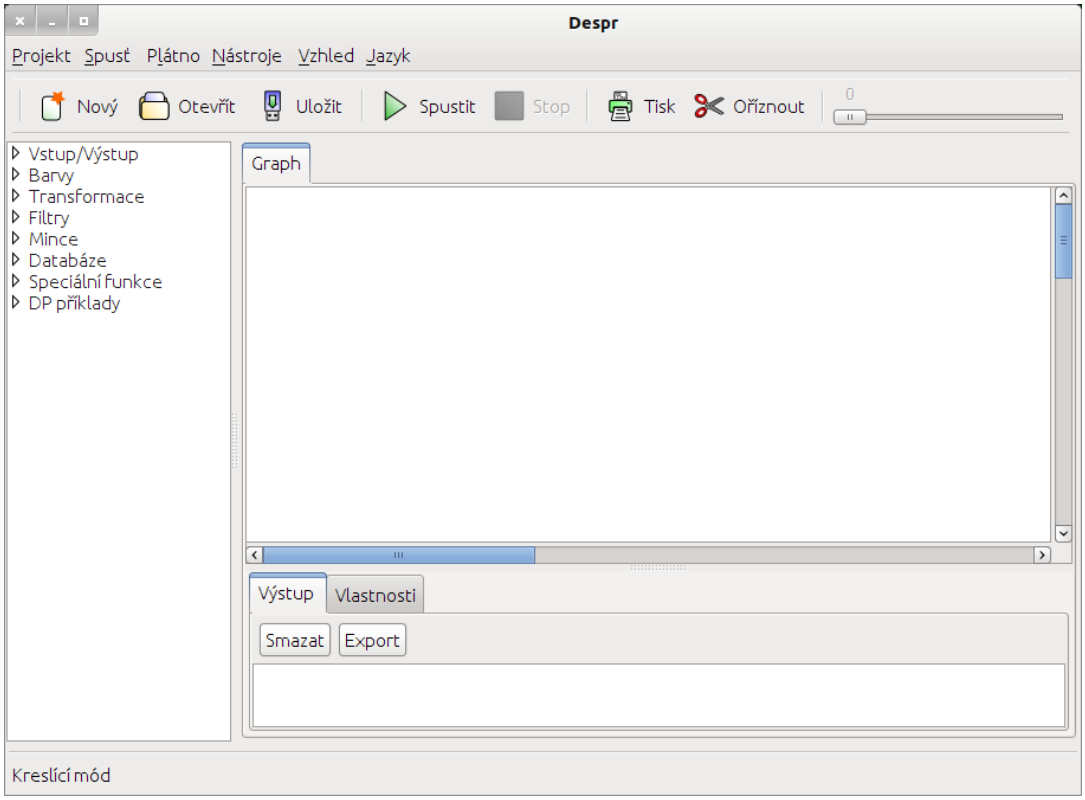
Obrázek 111: Aplikace spuštěna bez grafického uživatelského rozhraní.

### B.1.2 Standardní verze s GUI

Verze s grafickým rozhraním slouží hlavně pro vytváření a testování nových metod. Obsahuje plátno, na které je možné vkládat jednotlivé operace a spojovat je pomocí orientovaných hran do složitějších celků. Spouští se buď poklepáním myši na spustitelný balík (*Despr.jar*) nebo příkazem v příkazové řádce bez argumentů:

```
java -jar Despr.jar
```

Po spuštění se otevře okno, které bude vypadat podobně jako na obrázku 112, záleží totiž na použitém vzhledu a lokalizaci. Zde ukazovaná verze používá GTK+ nastavení s českou lokalizací.




Obrázek 112: Okno aplikace po spuštění v GUI.

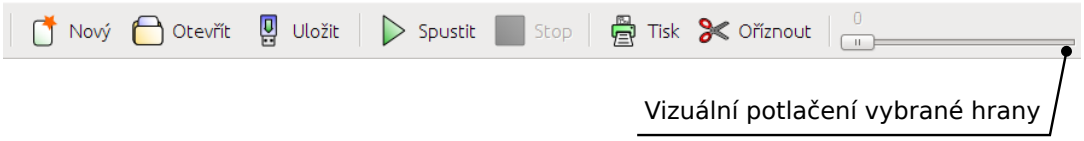
## B.2 Ovládání

Popis toho, jak se pracuje s operacemi a nastavování jejich parametrů, byl již popsán v textu, v kapitole popisující aplikaci, konkrétně kap. 4.4 a 4.5. Zde pouze ve stručnosti, operace se na plátno dostane, přetažením jejího názvu ze seznamu operací v levém panelu, na pracovní plochu. Kliknutím na operaci se vybere (to signalizuje světle modré orámování) a v tu chvíli se načtou i vnitřní parametry operace do panelu vlastností v dolní části aplikace.

Spojování operací se provádí kliknutím na některý z výstupních portů dané operace (porty ve spodní části), čímž se začne vykreslovat hrana. Kliknutím na plátno, se v daném místě hrana zalomí. Zmáčknutí pravého tlačítka myši, během kreslení hrany, přeruší tento proces. Kliknutím na kompatibilní vstupní port jiné operace je nová hrana přidána. Co jsou to kompatibilní porty, se je možné dočíst opět v textu. Nejběžnější rozlišení je podle barvy. Porty které vypadají vizuálně stejně, lze vždy propojit. Jak rozlišit „složitější“ situace je popsáno v kapitole 4.5.

### B.2.1 Menu a panel s rychle dostupnými nástroji

Téměř všechny funkce v panelu s rychle dostupnými nástroji se nachází v menu. Až na poslední, kterou je většinou nepřístupný posuvník (obr. 113). Ten je možné používat pouze v případě, že je vybrána některá z hran a slouží k jejímu vizuálnímu potlačení. Některé hrany v obraze, totiž mohou přenášet „méně důležité“ informace a aby nerušily je možné jim přidat jas až na hodnotu 230 (z 255), kdy hrana téměř splývá s pozadím a neruší tolik.




Obrázek 113: Lišta s rychle dostupnými nástroji.

**B.2.1.1 Menu** V této části jsou postupně vysvětleny jednotlivé položky v menu, jejich funkce a co dělají.

#### Projekt:

**Nový** [CTRL + N], vytvoří nový graf, resp. smaže obsah stávajícího plátna.

**Otevřít** [CTRL + O], načte graf z uloženého souboru.

**Uložit** [CTR + S], uloží aktuální graf i s nastavením vnitřních parametrů do souboru.

**Zavřítl** [ALT + F4], ukončí aplikaci.

#### Spusť:

**Spustit** [F5], spustí namodelovaný algoritmus.

**Stop** [ALT + ESC], ukončí započatý proces zpracování algoritmu.

**Kontrola grafu** [F6], provede kontrolu grafu, zda splňuje všechny náležitosti. Stejná kontrola se provádí vždy před spuštěním.

**Obnovit** [CTRL + ALT + R], vynuluje iterátory všech kořenových operací.

#### Plátno:

**Oříznout,** zmenší velikost plátna tak, aby byl vidět celý graf. Ale pouze v případě že plátno je větší než minimální velikost, která je nastavena na $1920 \times 1080$px.

**Tisk** [CTRL + ALT + P], vypíše graf v textové podobě do panelu výstup, kde je přesměrován, jak standardní výstup (stdout), tak ten chybový (stderr). Ve stejném formátu je vypsán graf i ve verzi bez GUI. Výpis vypadá tak, že jsou nejprve vypsány použité operace, vč. jejich nastavení a následně seznam jednotlivých hran.

### Nástroje:

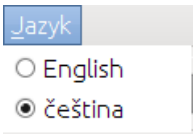
**Správce rozšíření,** otevře okno se správcem rozšíření. V něm je možné si přizpůsobit seznam operací, podle vlastní potřeby. Nahrávat nové operace, typy a typová rozšíření, vč. provázaní datového typu s jeho rozšířením. Tomu jak pracovat s tímto nástroje byla věnována celá kapitola 4.8 (*řešení rozšiřitelnosti*).

**Struktura použitých typů,** Nástroj slouží spíše jen pro zobrazení struktury aktuálně použitých datových typů, u portů operací. Navíc je pomocí něj možné měnit obarvení portů, například pokud by byla různým typům portů přidělena velmi podobná barva, což by mohlo být následně matoucí. Změnu barvy je možné dělat pouze u základních typů. Ty odvozené zdědí barvu od svých rodičovských typů.

**Vzhled:** Menu vzhled obsahuje seznam dostupných šablon v daném operačním systému. Pokud je při spuštění nastavena šablona, která není v rámci OS dostupná, je použit klasický javovský vzhled. V OS Ubuntu 11.10, který byl použitý pro vývoj a většinu testování, vypadá obsah menu jako na následujícím obrázku (obr. 114).


Obrázek 114: Menu: vzhled v OS Ubuntu 11.10.

**Jazyk:** Menu jazyk je taktéž generováno dynamicky, podle toho kolik existuje jazykových mutací. V době dokončení této práce jsou dostupné dvě jazykové mutace a to česká a anglická (obr. 115).




Obrázek 115: Menu: jazyk.

# C Programátorská dokumentace

Zde se nachází pouze popis toho, jak se výsledná aplikace a API sestavují. To z jakých tříd se nástroj skládá a jaké jsou vazby mezi nimi, bylo dostatečně podrobně popsáno v textu, kap 4. Nakonec je popsáno, jaké informace se ukládají do externích souborů, mimo aplikaci.

K sestavení aplikace byl napsaný buildovací skript pro ant. Jmenuje se `build-despr.xml` a nachází se v adresáři s aplikací. Spolu s ním jsou zde ještě dva soubory s uloženými parametry pro překlad, `despr-app.properies` a `despr-build.properties`. V prvním se nachází proměnné, které využívá skript pro sestavení aplikace, druhý je společný a využívá jej i skript sestavující API. Oba dva soubory připraví kompletní verze zabalené v zip archivu, včetně vygenerované javadoc dokumentace.

Pro sestavení rozšiřujících balíků, bylo využito vývojového prostředí Netbeans. Zde stačí pouze zabalit zkompilované soubory do společného `jar` balíku, který je aplikace schopna dále načíst. Všechna instalovaná rozšíření, vč. zdrojových souborů se nachází v adresáři `installed_extensions`.

## C.1 Sestavení aplikace

Pro sestavení aplikace je třeba spustit skript `build-despr.xml` v adresáři, kde se nachází složky:

- `src`,
- `lib`,
- `resources`,
- `extendions`.

Obsah výsledného archivu vypadá tak, jako v následujícím výpise. Po rozbalení je možné aplikaci rovnou spustit.

- `doc`,
- `extendions`,
- `lib`,
- `resources`,
- `src`,
- `Despr.jar`

## C.2 Obsah adresáře resources

Zde je popsán obsah adresáře s externími zdroji, s krátkým popiskem ke každému z nich.

**icons/** zde se nachází ikony použité v aplikaci.

**lang/** lokalizační soubory, pro každou lokalizaci se zde nachází podadresář s kódovým jménem. Lokalizace aplikace a operací jsou dvě různé věci. Zde se nachází lokalizační soubory pouze pro aplikaci. O lokalizaci operací se stará autor daného balíku a lokalizační soubory jsou publikovány v něm.

**available_extensions.properties** obsahuje seznam dostupných typových rozšíření. S tím že si u každého pamatuje počet, kolikrát již byl použit.

**ColorPalete.properties** seznam se základní paletou barev, které jsou vzájemně relativně kontrastní. Pokud je barva již použita nachází se u ní číslo jedna.

**despr.properties** zde se nachází informace o použitém vzhledu a jazykové mutaci. Je zde také seznam dostupných jazykových mutací. Pokud by chtěl někdo přidat další jazyk, musí jej zde dopsat.

**SavedColors.properties** Pro zajištění konzistence mezi různými běhy, si aplikace pamatuje, které barvy přidělila jakým typům. Přidělování nových barev, ještě k nepoužitým datovým typům u portů, probíhá automaticky a záleží na tom, který typ bude použit dříve. Kdyby si to aplikace nepamatovala, byla by barevnost portů, při každém spuštění jiná.

**extensions_of_types.xml** zde jsou uloženy informace o tom, s kterými datovými typy, jsou svázána, ta která typová rozšíření.

**Operations.xml** zde je uložena struktura stromu s operacemi, který se nachází v levém panelu v aplikaci.

**graph_model.xsd** pomocí tohoto souboru je kontrolována korektnost XML souboru popisujícího model uloženého grafu.

**graph_view.xsd** pomocí tohoto souboru je kontrolována korektnost XML souboru popisujícího pohledovou část uloženého grafu.

**unused_operations.list** zde se nachází seznam nepoužitých operací. To jsou operace, které se nachází v některých z rozšiřujících balíků, ale nejsou použity ve stromě operací.

# D Implementované operace

Tato část popisuje pouze operace, u kterých nemusí být zcela zřejmé, na první pohled, co dělají nebo co znamenají některé jejich parametry. Operace typu změna velikosti či převod do odstínu šedi, zde nebudou vysvětlovány. Také operace, jejichž princip byl již vysvětlen v textu zde nebudou, pokud tedy neobsahují parametry, jejichž použití by nemuselo být hned jasné.

**Načíst obrázky** jedná se o kořenovou operaci, která nenačítá jeden obrázek, ale všechny obrázky ve vstupní složce. Navíc obsahuje dva pomocné vstupní parametry, *rekurzivní průchod* a *seřadit data*. Pokud je první z nich nastaven na `true`, načtou se obrázky i ve všech podsložkách. Druhý parametr slouží pro setřízení dat či jejich zamíchání. Náhodný přistup k datům se hodil v některých menších testech a zůstal i ve finální operaci. Data jsou „míchána“ sice náhodně, ale pokaždé stejně náhodně. To je kvůli zajištění integrity mezi jednotlivými běhy. Na výstupu operace poskytne vždy jeden obrázek a informace o souboru, ve kterém se nachází. V dalším kole běhu je poskytnut druhý obrázek z kolekce, atd., dokud není vyčerpána celá kolekce. Vizuální podoba je na obrázku 116.

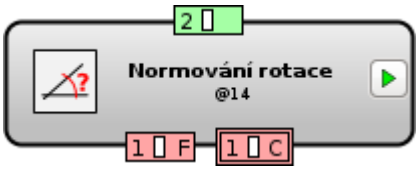

Obrázek 116: Načíst obrázky.

**Cannyho detektor hran** operace slouží pro detekci hran v obraze v odstínech šedi. Krom standardních parametrů, obsahuje také dva speciální parametry. První se jmenuje *doplnit strany?* a slouží pro doplnění stran na boku obrázku tak, že část obrázku je překopírována zleva doprava a část zprava doleva. Tato funkce se využívá hlavně u obrázků po polární transformaci, čímž je zajištěna spojitost přes osu $x$. Na obraz po polární transformaci se totiž pohlíží jako na pás, který je namotán na válci a boční strany jsou spojeny. Druhý parametr *barva pozadí*, představuje barvu, která je použita pro orámování obrazu. Velikost rámečku i toho jaká velká část bude překopírována ze stran, je odvozen od velikosti masky. Orámování řeší problém toho aby byla vždy použita celá maska a ne pouze její část na okrajích a v rozích. Barvu je vhodné volit tak aby korespondovala co nejvíce s barvou okolo okrajů obrázku. U mincí to je jednoduché, protože mají všechny bílé pozadí. Vizuální podoba operace je vidět na obrázku 117.


Obrázek 117: Načíst obrázky.

**Normování rotace a zjisti úhel z vektoru** obě dvě operace normují natočení obrazu. První z nich v sobě obsahuje všechny kroky, od převodu vstupního obrázku na vektor, přes vytvoření rozšířeného histogramu, až po výpočet normujícího úhlu tak, jak byla metoda popsána v aktuálním stavu projektu. Druhá operace, naopak očekává již před připravený histogram a provádí už jen určení úhlu, tj. rozřízne histogram v určené hladině a nalezne nejdelší spojitý úsek, ze kterého určí úhel. Na vstupu tak nemusí byt pouze rozšířený histogram, ale také histogram vyhlazený pouze Gaussovým filtrem nebo jinak upravený.

Obě dvě operace obsahují stejné vstupní parametry, a to procentuální vyjádření hladiny, ve které bude histogram rozdělen a prahovou hodnotu, která eliminuje příliš malé spojité úseky. Jejich vizuální podoby jsou vidět na obrázcích 118 a 119.




Obrázek 118: Normování rotace.

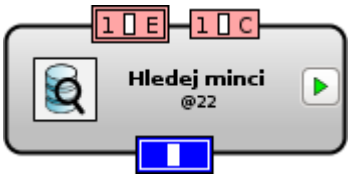

Obrázek 119: Zjisti úhel z vektoru.

**Binární obrázek na vektor** opreace vytvoří z binárního obrázku vektor celých čísel. Převod probíhá tak, že je v jednotlivých sloupečcích obrazu je spočítán počet bílých pixelů a toto je vráceno jako výstupní vektor. Operace se používá na obrázky po polární transformaci. Lze ji použít na jakýkoli jiný binární obrázek, ale už to nedává moc smysl. Operace je v aplikaci jen proto, že byla rozdělena původní operace *normování rotace* na několik úloh a tahle operace představuje první z nich. Také obsahuje jeden vstupní parametr, který urči jak se výsledné hodnoty vypočtou. Hodnota *histogram* spočítá četnost bílých pixelů v jednotlivých sloupečcích obrazu. Hodnota

*average* vypočte výslednou hodnotu vektoru, jako průměr z ypsilonových souřadnic. Poslední hodnota *median* funguje stejně jako metoda s hodnotou *average*, akorát místo průměru počítá střední hodnotu. Podoba operace je na obrázku 120.


Obrázek 120: Binární obrázek na vektor.

**Hledej minci** operace vyhledá minci, resp. poskytne seznam $N$ nejpodobnějších mincí. Na vstupu očekává vektor popisující minci a součet jeho položek. Další vnitřní parametry jsou standardní parametry pro připojení k databázi. **Metoda vyžaduje mít v databázi doinstalované metody `compute_distance` a `compare_vectors`.** Vizualizace operace je vidět na obrázku 121.




Obrázek 121: Hledej minci.

**Ulož nalezené mince** operace na základě informací získaných z databáze, překopíruje do vybrané složky nalezené obrázky. V databázi mince uloženy nejsou. Nachází se zde pouze adresa ke zdrojovému obrázku. Je tak třeba krom databáze mít mince, použité pro předzpracování, stále na stejném místě, jinak operace nebude fungovat. Mince jsou uloženy ve složce, tak že je zde obrázek s původním názvem, plus má prefix určující jeho pořadí. Pořadí jsou číslována od nuly. Dalším vstupem do operace je odkaz na obrázek mince, která byla vyhledávána. Pro ten je použit prefix s dvěma nulami. Posledním parametrem je jméno hledaného souboru, na jehož základě je vytvořen podadresář, do kterého jsou nakopírovány nalezené mince. Vizuální reprezentace operace je na obrázku 122.


Obrázek 122: Hledej minci.

**Ulož minci do databáze** tato operace naopak od té předchozí, uloží vytvořený popis mince (pomocí vektoru) do databáze, vč. dalších pomocných informací. Hlavními vstupy jsou vektor popisující minci, součet jeho položek, úhel který byl použit pro normování natočení[62], jméno souboru a cesta ke zdroji. Zbytek vnitřních parametrů slouží opět pro přípojení k databázi. Vizuální reprezentace operace je na obrázku 123.


Obrázek 123: Ulož minci do databáze.

**Vyobraz úhel v obraze po polární transformaci** operace pouze vyplní do obrazu po polární transformaci sloupec, který reprezentuje nalezený úhel, červenou barvou. Byla použita pro vizuální kontrolu toho, jak jednotlivé přístupy normovaly natočení. Jiné vyžití moc nemá. Vizuální podoba je na obrázku 124


Obrázek 124: Vyobraz úhel v obraze po polární transformaci.

**Převod na řetězec** toto je jediná operace v celé aplikaci, která očekává na vstupu typ `Object`. Ten je zajímavý tím, že na něj lze připojit jakoukoliv hranu. Operace jako taková nedělá nic těžkého, pouze zavolá na daný objekt metodu `toString`. Nicméně její použití se často hodí. Například pro převod čísla na řetězec, který je možné dále použít jako specifikaci jména nebo jméno podadresáře, pro metodu uložit obrázek.


Obrázek 125: Vyobraz úhel v obraze po polární transformaci.

[62] V případě metody, kdy úhel není počítán je možné port přepnout na vnitřní parametr a nastavit mu hodnotu 1.

## D.1 Instalace rozšiřujících funkcí do MySQL databáze.

Zdrojový soubor potřebný pro rozšíření databáze, o možnost porovnávání dvou vektorů, se nachází ve zdrojích, v archivu uchovávajícím zdrojové soubory k operacím spolupracujícími s databází (`Despr_DatabaseOperations.zip`). V archivu se nachází adresář c_src, kde je jak zdrojový soubor tak přeložená sdílená knihovna `vector_operations.so`.

### D.1.1 Postup instalace rozšíření

Tímto postupem byly přidány funkce do MySQL databáze nainstalované na počítači s OS Ubuntu 11.10. V MS Windows byla testována pouze funkčnost aplikace jako takové. Databáze se pokaždé nacházela na počítači s linuxem a všechny testy taktéž probíhaly na linuxovém stroji. Pokud bude uživatel chtít operaci použít na jiném stroji, musí zajistit, aby měl příslušné metody v databázi doinstalované, jinak metoda nebude fungovat.

1. Překlad:

```
$ gcc - shared -o vector_operations.so vector_oprations.c
-I /usr/include/mysql -lm
```

2. Přidání knihovny do MySQL adresáře:

```
$ sudo cp vector_operations.so /usr/lib/mysql/plugins
```

3. Přidání funkcí do v databázi:

```
> create function compute_distance returns real soname
"vector_operations.so"
```

```
> create function compare_vectors returns integer soname
"vector_operations.so"
```

# E Příklady s ukládáním grafů

Tato příloha popisuje složitější, resp. rozsáhlejší příklady uložení grafu. První příklad popisuje ukládání komplexních datových typů. V druhém jsou pak pouze výpisy XML struktur popisující graf na obrázku 57. Ty z důvodu rozsáhlosti XML souborů byly přesunuty zde do přílohy.

## E.1 Příklad uložení komplexního datového typu

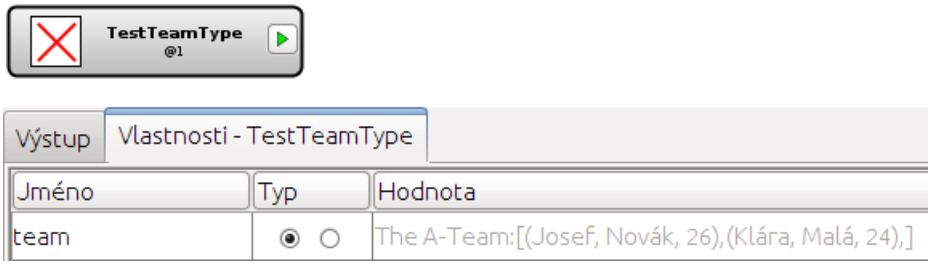
Tento příklad pouze ukazuje, že nejsou prakticky kladena omezení na datové typy vstupních parametrů, pro to aby mohly být jejich hodnoty uloženy a znovu načteny. Operace `TestTeamType` slouží jen pro demonstraci, v aplikaci použít nelze použít, jelikož prakticky nic nedělá a nemá ani vstupní a výstupní porty. Kód operace ukazuje následující výpis (výpis. 10).

```
public class TestTeamType implements IOperation {

    public TestTeamType() {
        team = new Team();
        team.setName("The A-Team");
        Person p1 = new Person("Josef", "Novak", ESex.MALE, 26, new Contact(
                "josef.novak@ateam.cz", "123456789"));
        team.addPerson(p1);
        Person p2 = new Person("Klara", "Mala", ESex.FEMALE, 24, new Contact(
                "klara.mala@ateam.cz", null));
        team.addPerson(p2);
    }

    @AInputParameter(EInputParameterType.INNER)
    private Team team;

    @Override
    public void execute() throws Exception { }

    @Override
    public String getLocalizeMessage(String string) { return null; }

    public Team getTeam() { return team; }
    public void setTeam(Team team) { this.team = team; }
}
```

Výpis 10: Kód operace `TestTeamType`.

Jak je z kódu vidět operace pouze definuje jeden vnitřní parametr typy `Team` a v konstruktoru jej naplní. Ve skutečnosti takto komplexní vnitřní parametry v operacích příliš často nebudou, protože čím je datový typ složitější, tím musí být i editor jeho hodnoty složitější. V reálných operacích se většinou jedná právě o primitivní typy nebo o výčty. Pro zajímavost je na obrázku 126 vidět vizuální reprezentace operace v aplikaci.

V obrázku si je možné povšimnout, že hodnota parametru `team` je prošedivělá. Je to dáno tím, že není definován pro typ `Team` editor (`ParameterCellEditor`), pomocí




Obrázek 126: Vizuální prezentace operace, vč. tabulky s vlastnostmi.

nějž by bylo možné hodnotu upravit. Následující výpisy (výpisy 11 až 12) ukáží, z čeho všeho se typ `Team` skládá.

```
public class Team {

    private String name;
    private List<Person> people;

    public Team() {
        int teamIDGenerator = ID.createNewIDGenerator();
        name = "team_" + ID.getNextID(teamIDGenerator);
        people = new ArrayList<Person>();
    }

    public void addPerson(Person person) { people.add(person); }
    public void removePerson(Person person) { people.remove(person); }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public List<Person> getPeople() { return people; }
    public void setPeople(List<Person> people) { this.people = people; }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("%s:[", name));
        for (Person p : people) {
            sb.append(String.format("%s,", p.toString()));
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Výpis 11: Kód třídy `Team`.

Tým který představuje třída `Team` (výpis 11) se skládá z jména týmu a seznamu osob, které se v něm nachází. Jeden člověk je reprezentován pomocí třídy `Person` (výpis 12). Ta uchovává jméno, příjmení, pohlaví, věk a kontakt. Kontakt je také uložen ve vlastním

typy `Contact` (výpis 13), který umožňuje uchovat telefonní číslo a email. Pohlaví je řešeno pomocí výčtového typy `ESex`, které obsahuje pouze položky `MALE` a `FEMALE`.

```
public class Person {

    private String name;
    private String lastname;
    private ESex sex;
    private int age;
    private Contact contact;

    public Person() { }
    public Person(String name, String lastname, ESex sex, int age, Contact contact) {
        this.name = name;
        this.lastname = lastname;
        this.sex = sex;
        this.age = age;
        this.contact = contact;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getLastname() { return lastname; }
    public void setLastname(String lastname) { this.lastname = lastname; }
    public ESex getSex() { return sex; }
    public void setSex(ESex sex) { this.sex = sex; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public Contact getContact() { return contact; }
    public void setContact(Contact contact) { this.contact = contact; }

   @Override
    public String toString() { return String.format("(%s, %s, %d)", name, lastname, age); }
}
```

Výpis 12: Kód třídy `Person`.

```
public class Contact {

    private String email;
    private String phoneNumber;

    public Contact() { }
    public Contact(String email, String phoneNumber) {
        this.email = email;
        this.phoneNumber = phoneNumber;
    }
    public String getEmail() { return email;}
    public void setEmail(String email) { this.email = email;}
    public String getPhoneNumber() { return phoneNumber; }
    public void setPhoneNumber(String phoneNumber) { this.phoneNumber = phoneNumber; }
}
```

Výpis 13: Kód třídy `Contact`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graph>
<operations>
  <operation id="1" root="false" type="cz.vsb.cs.sur096.despr.operations.examples.TestTeamType">
      <object name="team" type="cz.vsb.cs.sur096.despr.types.Team">
          <primitive name="name">
              <string>The A-Team</string>
          </primitive>
          <collection name="people" type="java.util.ArrayList">
              <item index="0">
                  <object name="" type="cz.vsb.cs.sur096.despr.types.Person">
                      <enum name="sex" type="cz.vsb.cs.sur096.despr.types.ESex">MALE</enum>
                      <primitive name="age">
                          <int>26</int>
                      </primitive>
                      <primitive name="name">
                          <string>Josef</string>
                      </primitive>
                      <primitive name="lastname">
                          <string>Novak</string>
                      </primitive>
                      <object name="contact" type="cz.vsb.cs.sur096.despr.types.Contact">
                          <primitive name="phoneNumber">
                              <string>123456789</string>
                          </primitive>
                          <primitive name="email">
                              <string>josef.novak@ateam.cz</string>
                          </primitive>
                      </object>
                  </object>
              </item>
              <item index="1">
                  <object name="" type="cz.vsb.cs.sur096.despr.types.Person">
                      <enum name="sex" type="cz.vsb.cs.sur096.despr.types.ESex">FEMALE</enum>
                      <primitive name="age">
                          <int>24</int>
                      </primitive>
                      <primitive name="name">
                          <string>Klara</string>
                      </primitive>
                      <primitive name="lastname">
                          <string>Mala</string>
                      </primitive>
                      <object name="contact" type="cz.vsb.cs.sur096.despr.types.Contact">
                          <null_object name="phoneNumber" type="java.lang.String"/>
                          <primitive name="email">
                              <string>klara.mala@ateam.cz</string>
                          </primitive>
                      </object>
                  </object>
              </item>
          </collection>
      </object>
  </operation>
</operations>
<edges/>
</graph>
```

Výpis 14: XML soubor představující model uloženého grafu s operací `TestTeamType`.

Poslední výpis (výpis 14) ukazuje, jak vypadá model uloženého grafu s operací, která obsahuje, možná až příliš složitý, datový typ `Team` jako vnitřní parametr. Nicméně je vidět, že meze fantazie se autorům operací nekladou. Pouze při používání vlastních typů jako vnitřních parametrů operací je třeba dodržet pár pravidel. Na druhou stranu je třeba složitost typů udržet na rozumné míře. Přeci jen, když bude operace obsahovat více podobně složitých vnitřních parametrů, nebude operace jako taková moc použitelná. Protože uživatelé ji nebudou chtít složitě nastavovat. V případě, kdy vstupní parametr je příliš složitý, je lepší ho raději uzamknout jak vstupní port. Jeho hodnotu pak bude muset vygenerovat jiná operace a uživateli je vcelku jedno jak je typ složitý. Navíc takové parametry nejsou ani ukládány a není třeba, aby jejich datové typy splňovali výše jmenované podmínky.

## E.2 XML soubory vztahující se k příkladu na obrázku 57

Zde se nachází pouze výpisy souborů `model.xml` (výpis 15) a `view.xml` (výpis 16) náležející k danému příkladu.

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<graph>
    <operations>
        <operation id="1" root="true" type="cz.vsb.cs.sur096.despr.operations.io.LoadImages">
            <object name="srcDir" type="cz.vsb.cs.sur096.despr.types.Directory">
                <object name="file" type="java.io.File">
                    <primitive name="path">
                        <string>/home/sur096/test1/0_resize/11</string>
                    </primitive>
                </object>
            </object>
            <primitive name="recursiveBrowse">
                <bool>false</bool>
            </primitive>
            <primitive name="shuffle">
                <bool>false</bool>
            </primitive>
        </operation>
        <operation id="2" root="false" type="cz.vsb.cs.sur096.despr.operations.io.FileExtractor"/>
        <operation id="3" root="false" type="cz.vsb.cs.sur096.despr.operations.io.SaveImage">
            <object name="destDir" type="cz.vsb.cs.sur096.despr.types.Directory">
                <object name="file" type="java.io.File">
                    <primitive name="path">
                        <string>/home/sur096/A_TEST_IMG/11</string>
                    </primitive>
                </object>
            </object>
            <primitive name="subDirPath">
                <string/>
            </primitive>
            <primitive name="nameSpecification">
                <string/>
            </primitive>
        </operation>
    </operations>
    <edges>
        <edge id="9">
            <source operation_id="1">file</source>
            <target operation_id="2">file</target>
```

```
        </edge>
        <edge id="10">
            <source operation_id="2">fileName</source>
            <target operation_id="3">fileName</target>
        </edge>
        <edge id="12">
            <source operation_id="1">outImg</source>
            <target operation_id="3">inputImg</target>
        </edge>
    </edges>
</graph>
```

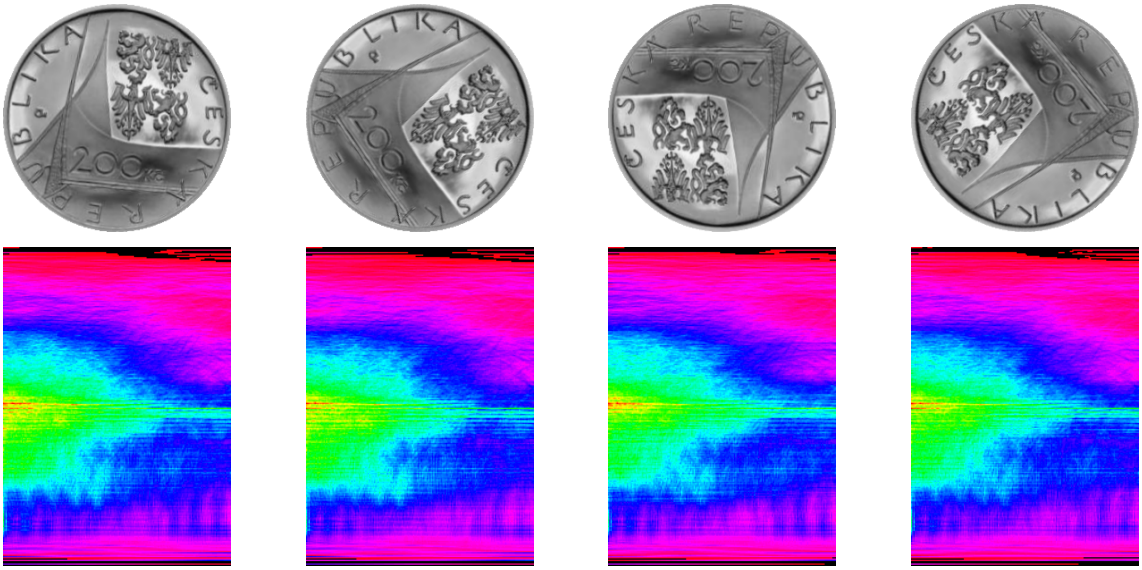
Výpis 15: Struktura XML souboru popisujícího model grafu.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graph>
    <operations>
        <operation id="1">
            <position>
                <x>219</x>
                <y>0</y>
            </position>
        </operation>
        <operation id="2">
            <position>
                <x>304</x>
                <y>113</y>
            </position>
        </operation>
        <operation id="3">
            <position>
                <x>216</x>
                <y>257</y>
            </position>
        </operation>
    </operations>
    <edges>
        <edge color="0" id="9">
            <point id="5">
                <x>334</x>
                <y>71</y>
            </point>
            <input_port_bounds id="6">
                <x>390</x>
                <y>118</y>
                <width>35</width>
                <height>15</height>
            </input_port_bounds>
        </edge>
        <edge color="0" id="10">
            <point id="8">
                <x>384</x>
                <y>186</y>
            </point>
            <input_port_bounds id="15">
                <x>312</x>
                <y>262</y>
                <width>35</width>
                <height>15</height>
            </input_port_bounds>
        </edge>
        <edge color="0" id="12">
```

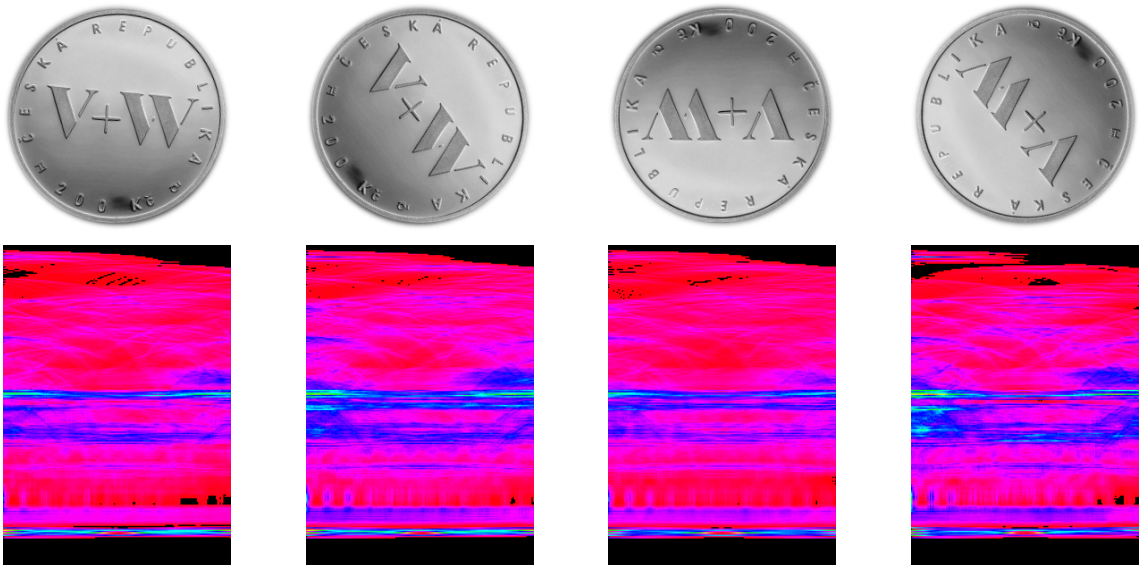
```
            <point id="4">
                <x>289</x>
                <y>71</y>
            </point>
            <point id="30">
                <x>247</x>
                <y>168</y>
            </point>
            <input_port_bounds id="14">
                <x>267</x>
                <y>262</y>
                <width>35</width>
                <height>15</height>
            </input_port_bounds>
        </edge>
    </edges>
</graph>
```

Výpis 16: Struktura XML souboru popisujícího grafickou reprezentaci grafu.

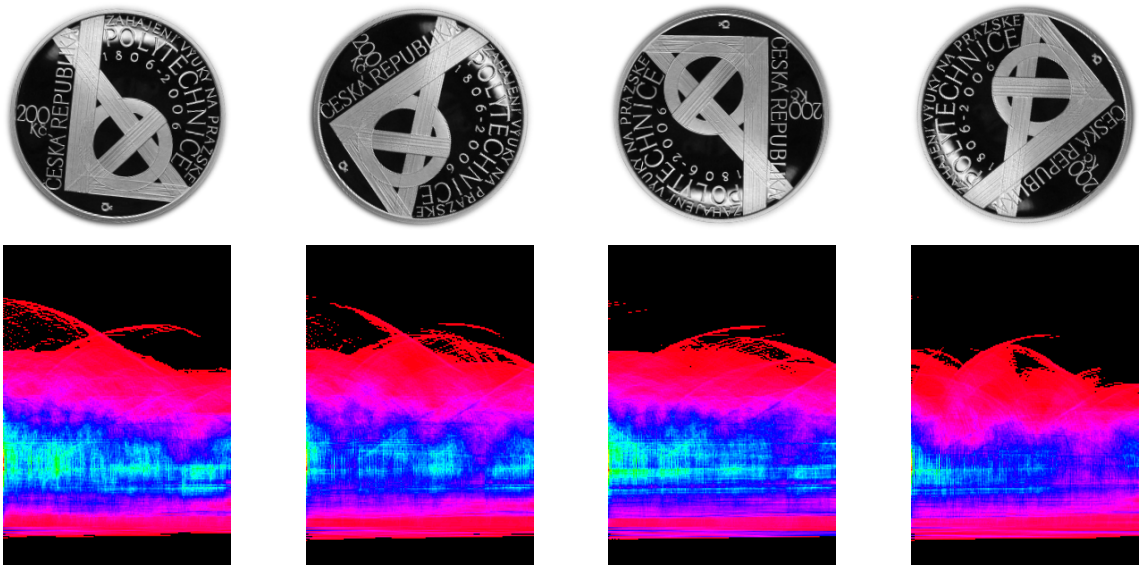
# F Ukázky několika výstupů metody popisující obraz nezávisle na rotací.



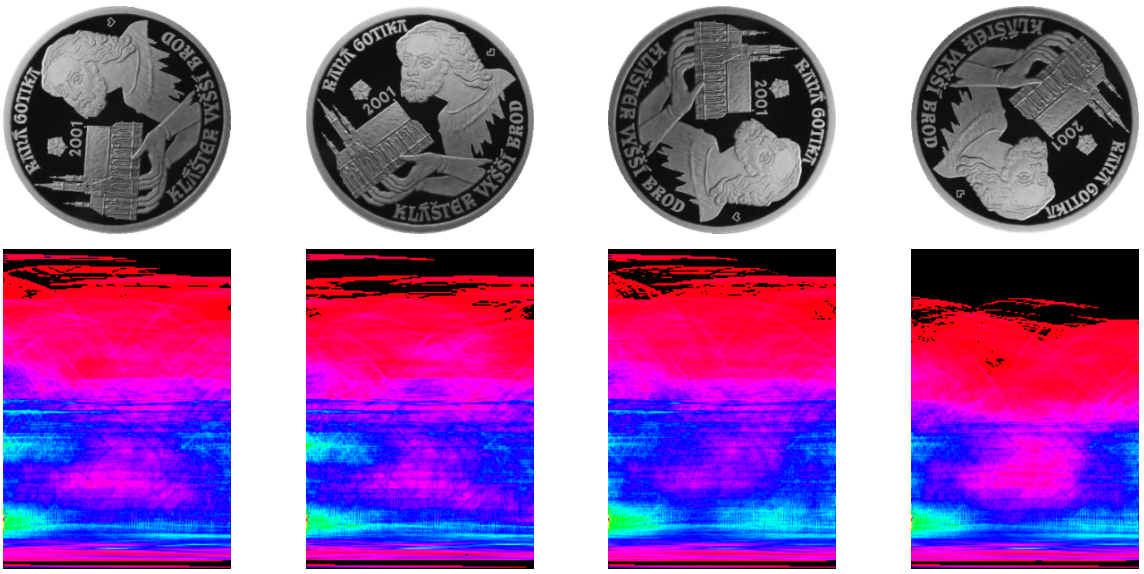
Obrázek 127: Ukázka 1 – vizualizace otisků mince, vypočtených pro natočení: 6°, 56°, 181° a 231°.



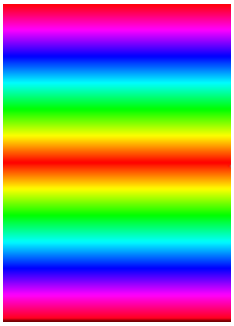
Obrázek 128: Ukázka 2 – vizualizace otisků mince, vypočtených pro natočení: 6°, 56°, 181° a 231°.



Obrázek 129: Ukázka 3 – vizualizace otisků mince, vypočtených pro natočení: 6°, 56°, 181° a 231°.



Obrázek 130: Ukázka 4 – vizualizace otisků mince, vypočtených pro natočení: 6°, 56°, 181° a 231°.



Obrázek 131: Výsledek metody aplikované na bílý obraz o rozměrech $127 \times 360$px.